Node.js v9.11.2 Documentation

Index | View on single page | View as JSON | View another version ▾

Table of Contents

- buf.equals(otherBuffer)
- buf.fill(value[, offset[, end]][, encoding])
- buf.includes(value[, byteOffset][, encoding])
- buf.indexOf(value[, byteOffset][, encoding])
- buf.keys()
- buf.lastIndexOf(value[, byteOffset][, encoding])
- buf.length
- buf.parent deprecated
- buf.readDoubleBE(offset[, noAssert])
- buf.readDoubleLE(offset[, noAssert])
- buf.readFloatBE(offset[, noAssert])
- buf.readFloatLE(offset[, noAssert])
- buf.readInt8(offset[, noAssert])
- buf.readInt16BE(offset[, noAssert])
- buf.readInt16LE(offset[, noAssert])
- buf.readInt32BE(offset[, noAssert])
- buf.readInt32LE(offset[, noAssert])
- buf.readIntBE(offset, byteLength[, noAssert])
- buf.readIntLE(offset, byteLength[, noAssert])
- buf.readUInt8(offset[, noAssert])
- buf.readUInt16BE(offset[, noAssert])
- buf.readUInt16LE(offset[, noAssert])
- buf.readUInt32BE(offset[, noAssert])
- buf.readUInt32LE(offset[, noAssert])
- buf.readUIntBE(offset, byteLength[, noAssert])
- buf.readUIntLE(offset, byteLength[, noAssert])
- buf.slice([start[, end]])
- buf.swap16()
- buf.swap32()
- buf.swap64()
- buf.toJSON()
- buf.toString([encoding[, start[, end]]])
- buf.values()
- buf.write(string[, offset[, length]][, encoding])
- buf.writeDoubleBE(value, offset[, noAssert])
- buf.writeDoubleLE(value, offset[, noAssert])
- buf.writeFloatBE(value, offset[, noAssert])
- buf.writeFloatLE(value, offset[, noAssert])
- buf.writeInt8(value, offset[, noAssert])
- buf.writeInt16BE(value, offset[, noAssert])
- buf.writeInt16LE(value, offset[, noAssert])
- buf.writeInt32BE(value, offset[, noAssert])
- buf.writeInt32LE(value, offset[, noAssert])
- buf.writeIntBE(value, offset, byteLength[, noAssert])
- buf.writeIntLE(value, offset, byteLength[, noAssert])
- buf.writeUInt8(value, offset[, noAssert])
- buf.writeUInt16BE(value, offset[, noAssert])
- buf.writeUInt16LE(value, offset[, noAssert])
- buf.writeUInt32BE(value, offset[, noAssert])
- buf.writeUInt32LE(value, offset[, noAssert])

- ERR_ENCODING_NOT_SUPPORTED
- ERR_FALSY_VALUE_REJECTION
- ERR_HTTP_HEADERS_SENT
- ERR_HTTP_INVALID_STATUS_CODE
- ERR_HTTP_TRAILER_INVALID
- ERR_HTTP2_ALTSVC_INVALID_ORIGIN
- ERR_HTTP2_ALTSVC_LENGTH
- ERR_HTTP2_CONNECT_AUTHORITY
- ERR_HTTP2_CONNECT_PATH
- ERR_HTTP2_CONNECT_SCHEME
- ERR_HTTP2_GOAWAY_SESSION
- ERR_HTTP2_HEADER_SINGLE_VALUE
- ERR_HTTP2_HEADERS_AFTER_RESPOND
- ERR_HTTP2_HEADERS_SENT
- ERR_HTTP2_INFO_STATUS_NOT_ALLOWED
- ERR_HTTP2_INVALID_CONNECTION_HEADERS
- ERR_HTTP2_INVALID_HEADER_VALUE
- ERR_HTTP2_INVALID_INFO_STATUS
- ERR_HTTP2_INVALID_PACKED_SETTINGS_LENGTH
- ERR_HTTP2_INVALID_PSEUDOHEADER
- ERR_HTTP2_INVALID_SESSION
- ERR_HTTP2_INVALID_SETTING_VALUE
- ERR_HTTP2_INVALID_STREAM
- ERR_HTTP2_MAX_PENDING_SETTINGS_ACK
- ERR_HTTP2_NO_SOCKET_MANIPULATION
- ERR_HTTP2_OUT_OF_STREAMS
- ERR_HTTP2_PAYLOAD_FORBIDDEN
- ERR_HTTP2_PING_CANCEL
- ERR_HTTP2_PING_LENGTH
- ERR_HTTP2_PSEUDOHEADER_NOT_ALLOWED
- ERR_HTTP2_PUSH_DISABLED
- ERR_HTTP2_SEND_FILE
- ERR_HTTP2_SESSION_ERROR
- ERR_HTTP2_SOCKET_BOUND
- ERR_HTTP2_STATUS_101
- ERR_HTTP2_STATUS_INVALID
- ERR_HTTP2_STREAM_CANCEL
- ERR_HTTP2_STREAM_ERROR
- ERR_HTTP2_STREAM_SELF_DEPENDENCY
- ERR_HTTP2_UNSUPPORTED_PROTOCOL
- ERR_INDEX_OUT_OF_RANGE
- ERR_INSPECTOR_ALREADY_CONNECTED
- ERR_INSPECTOR_CLOSED
- ERR_INSPECTOR_NOT_AVAILABLE
- ERR_INSPECTOR_NOT_CONNECTED
- ERR_INVALID_ARG_TYPE
- ERR_INVALID_ARG_VALUE
- ERR_INVALID_ARRAY_LENGTH
- ERR_INVALID_ASYNC_ID
- ERR_INVALID_BUFFER_SIZE

- ERR_INVALID_CALLBACK
- ERR_INVALID_CHAR
- ERR_INVALID_CURSOR_POS
- ERR_INVALID_DOMAIN_NAME
- ERR_INVALID_FD
- ERR_INVALID_FD_TYPE
- ERR_INVALID_FILE_URL_HOST
- ERR_INVALID_FILE_URL_PATH
- ERR_INVALID_HANDLE_TYPE
- ERR_INVALID_HTTP_TOKEN
- ERR_INVALID_IP_ADDRESS
- ERR_INVALID_OPT_VALUE
- ERR_INVALID_OPT_VALUE_ENCODING
- ERR_INVALID_PERFORMANCE_MARK
- ERR_INVALID_PROTOCOL
- ERR_INVALID_REPL_EVAL_CONFIG
- ERR_INVALID_SYNC_FORK_INPUT
- ERR_INVALID_THIS
- ERR_INVALID_TUPLE
- ERR_INVALID_URI
- ERR_INVALID_URL
- ERR_INVALID_URL_SCHEME
- ERR_IPC_CHANNEL_CLOSED
- ERR_IPC_DISCONNECTED
- ERR_IPC_ONE_PIPE
- ERR_IPC_SYNC_FORK
- ERR_METHOD_NOT_IMPLEMENTED
- ERR_MISSING_ARGS
- ERR_MISSING_MODULE
- ERR_MODULE_RESOLUTION_LEGACY
- ERR_MULTIPLE_CALLBACK
- ERR_NAPI_CONS_FUNCTION
- ERR_NAPI_INVALID_DATAVIEW_ARGS
- ERR_NAPI_INVALID_TYPEDARRAY_ALIGNMENT
- ERR_NAPI_INVALID_TYPEDARRAY_LENGTH
- ERR_NO_CRYPTO
- ERR_NO_ICU
- ERR_NO_LONGER_SUPPORTED
- ERR_OUT_OF_RANGE
- ERR_REQUIRE_ESM
- ERR_SERVER_ALREADY_LISTEN
- ERR_SOCKET_ALREADY_BOUND
- ERR_SOCKET_BAD_BUFFER_SIZE
- ERR_SOCKET_BAD_PORT
- ERR_SOCKET_BAD_TYPE
- ERR_SOCKET_BUFFER_SIZE
- ERR_SOCKET_CANNOT_SEND
- ERR_SOCKET_CLOSED
- ERR_SOCKET_DGRAM_NOT_RUNNING
- ERR_STDERR_CLOSE

    - Event: 'connect'
    - Event: 'connection'
    - Event: 'request'
    - Event: 'upgrade'
    - server.close([callback])
    - server.listen()
    - server.listening
    - server.maxHeadersCount
    - server.setTimeout([msecs][, callback])
    - server.timeout
    - server.keepAliveTimeout
- Class: http.ServerResponse
    - Event: 'close'
    - Event: 'finish'
    - response.addTrailers(headers)
    - response.connection
    - response.end([data][, encoding][, callback])
    - response.finished
    - response.getHeader(name)
    - response.getHeaderNames()
    - response.getHeaders()
    - response.hasHeader(name)
    - response.headersSent
    - response.removeHeader(name)
    - response.sendDate
    - response.setHeader(name, value)
    - response.setTimeout(msecs[, callback])
    - response.socket
    - response.statusCode
    - response.statusMessage
    - response.write(chunk[, encoding][, callback])
    - response.writeContinue()
    - response.writeHead(statusCode[, statusMessage][, headers])
- Class: http.IncomingMessage
    - Event: 'aborted'
    - Event: 'close'
    - message.destroy([error])
    - message.headers
    - message.httpVersion
    - message.method
    - message.rawHeaders
    - message.rawTrailers
    - message.setTimeout(msecs, callback)
    - message.socket
    - message.statusCode
    - message.statusMessage
    - message.trailers
    - message.url
- http.METHODS
- http.STATUS_CODES

About this Documentation

The goal of this documentation is to comprehensively explain the Node.js API, both from a reference as well as a conceptual point of view. Each section describes a built-in module or high-level concept.

Where appropriate, property types, method arguments, and the arguments provided to event handlers are detailed in a list underneath the topic heading.

Contributing

If errors are found in this documentation, please submit an issue or see the contributing guide for directions on how to submit a patch.

Every file is generated based on the corresponding `.md` file in the `doc/api/` folder in Node.js's source tree. The documentation is generated using the `tools/doc/generate.js` program. An HTML template is located at `doc/template.html`.

Stability Index

Throughout the documentation are indications of a section's stability. The Node.js API is still somewhat changing, and as it matures, certain parts are more reliable than others. Some are so proven, and so relied upon, that they are unlikely to ever change at all. Others are brand new and experimental, or known to be hazardous and in the process of being redesigned.

The stability indices are as follows:

Stability: 0 - Deprecated This feature is known to be problematic, and changes may be planned. Do not rely on it. Use of the feature may cause warnings to be emitted. Backwards compatibility across major versions should not be expected.

Stability: 1 - Experimental This feature is still under active development and subject to non-backwards compatible changes, or even removal, in any future version. Use of the feature is not recommended in production environments. Experimental features are not subject to the Node.js Semantic Versioning model.

Stability: 2 - Stable The API has proven satisfactory. Compatibility with the npm ecosystem is a high priority, and will not be broken unless absolutely necessary.

Caution must be used when making use of `Experimental` features, particularly within modules that may be used as dependencies (or dependencies of dependencies) within a Node.js application. End users may not be aware that experimental features are being used, and therefore may experience unexpected failures or behavior changes when API modifications occur. To help avoid such surprises, `Experimental` features may require a command-line flag to explicitly enable them, or may cause a process warning to be emitted. By default, such warnings are printed to `stderr` and may be handled by attaching a listener to the `process.on('warning')` event.

JSON Output

Added in: v0.6.12

Stability: 1 - Experimental

Every `.html` document has a corresponding `.json` document presenting the same information in a structured manner. This feature is experimental, and added for the benefit of IDEs and other utilities that wish to do programmatic things with the documentation.

Syscalls and man pages

System calls like open(2) and read(2) define the interface between user programs and the underlying operating system. Node functions which simply wrap a syscall, like `fs.open()`, will document that. The docs link to the corresponding man pages (short for manual pages) which describe how the syscalls work.

Some syscalls, like lchown(2), are BSD-specific. That means, for example, that `fs.lchown()` only works on macOS and other BSD-derived systems, and is not available on Linux.

Most Unix syscalls have Windows equivalents, but behavior may differ on Windows relative to Linux and macOS. For an example of the subtle ways in which it's sometimes impossible to replace Unix syscall semantics on Windows, see Node issue 4760.

Usage

`node [options] [V8 options] [script.js | -e "script" | - ] [arguments]`

Please see the Command Line Options document for information about different options and ways to run scripts with Node.js.

Example

An example of a web server written with Node.js which responds with `'Hello World!'`:

Commands displayed in this document are shown starting with `$` or `>` to replicate how they would appear in a user's terminal. Do not

include the `$` and `>` character they are there to indicate the start of each command.

There are many tutorials and examples that follow this convention: `$` or `>` for commands run as a regular user, and `#` for commands that should be executed as an administrator.

Lines that don't start with `$` or `>` character are typically showing the output of the previous command.

Firstly, make sure to have downloaded and installed Node.js. See this guide for further install information.

Now, create an empty project folder called `projects`, navigate into it: Project folder can be named base on user's current project title but this example will use `projects` as the project folder.

Linux and Mac:

```
$ mkdir ~/projects
$ cd ~/projects
```

Windows CMD:

```
> mkdir %USERPROFILE%\projects
> cd %USERPROFILE%\projects
```

Windows PowerShell:

```
> mkdir $env:USERPROFILE\projects
> cd $env:USERPROFILE\projects
```

Next, create a new source file in the `projects` folder and call it `hello-world.js`.

In Node.js it is considered good style to use hyphens (`-`) or underscores (`_`) to separate multiple words in filenames.

Open `hello-world.js` in any preferred text editor and paste in the following content.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Save the file, go back to the terminal window enter the following command:

```
$ node hello-world.js
```

An output like this should appear in the terminal to indicate Node.js server is running:

```
 Server running at http://127.0.0.1:3000/
 `
```

Now, open any preferred web browser and visit `http://127.0.0.1:3000`.

If the browser displays the string `Hello, world!`, that indicates the server is working.

Many of the examples in the documentation can be run similarly.

Assert

Stability: 2 - Stable

The `assert` module provides a simple set of assertion tests that can be used to test invariants.

A `strict` and a `legacy` mode exist, while it is recommended to only use strict mode.

For more information about the used equality comparisons see MDN's guide on equality comparisons and sameness.

Strict mode

▸ History

When using the `strict mode`, any `assert` function will use the equality used in the strict function mode. So `assert.deepEqual()` will, for example, work the same as `assert.deepStrictEqual()`.

On top of that, error messages which involve objects produce an error diff instead of displaying both objects. That is not the case for the legacy mode.

It can be accessed using:

```
const assert = require('assert').strict;
```

Example error diff (the `expected`, `actual`, and `Lines skipped` will be on a single row):

```
const assert = require('assert').strict;

assert.deepEqual([[[1, 2, 3]], 4, 5], [[[1, 2, '3']], 4, 5]);
```

```
AssertionError [ERR_ASSERTION]: Input A expected to deepStrictEqual input B:
+ expected
- actual
... Lines skipped

  [
    [
...
      2,
-     3
+     '3'
    ],
...
    5
  ]
```

To deactivate the colors, use the `NODE_DISABLE_COLORS` environment variable. Please note that this will also deactivate the colors in the REPL.

Legacy mode

Stability: 0 - Deprecated: Use strict mode instead.

When accessing `assert` directly instead of using the `strict` property, the Abstract Equality Comparison will be used for any function without "strict" in its name, such as `assert.deepEqual()`.

It can be accessed using:

```
const assert = require('assert');
```

It is recommended to use the `strict mode` instead as the Abstract Equality Comparison can often have surprising results. This is especially true for `assert.deepEqual()`, where the comparison rules are lax:

```
// WARNING: This does not throw an AssertionError!
assert.deepEqual(/a/gi, new Date());
```

assert(value[, message])

Added in: v0.5.9

- `value` <any>
- `message` <any>

An alias of `assert.ok()`.

assert.deepEqual(actual, expected[, message])

▶ History

- `actual` <any>
- `expected` <any>
- `message` <any>

Strict mode

An alias of `assert.deepStrictEqual()`.

Legacy mode

Stability: 0 - Deprecated: Use `assert.deepStrictEqual()` instead.

Tests for deep equality between the `actual` and `expected` parameters. Primitive values are compared with the Abstract Equality Comparison (`==`).

Only enumerable "own" properties are considered. The `assert.deepEqual()` implementation does not test the `[[Prototype]]` of objects or enumerable own `Symbol` properties. For such checks, consider using `assert.deepStrictEqual()` instead. `assert.deepEqual()` can have potentially surprising results. The following example does not throw an `AssertionError` because the properties on the `RegExp` object are not enumerable:

```
// WARNING: This does not throw an AssertionError!
assert.deepEqual(/a/gi, new Date());
```

An exception is made for `Map` and `Set`. Maps and Sets have their contained items compared too, as expected.

"Deep" equality means that the enumerable "own" properties of child objects are evaluated also:

```
const assert = require('assert');

const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    b: 2
  }
};
const obj3 = {
  a: {
    b: 1
  }
};
const obj4 = Object.create(obj1);

assert.deepEqual(obj1, obj1);
// OK, object is equal to itself

assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }
// values of b are different

assert.deepEqual(obj1, obj3);
// OK, objects are equal

assert.deepEqual(obj1, obj4);
// AssertionError: { a: { b: 1 } } deepEqual {}
// Prototypes are ignored
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an Error then it will be thrown instead of the `AssertionError`.

assert.deepStrictEqual(actual, expected[, message])

▸ History

- `actual` <any>
- `expected` <any>
- `message` <any>

Tests for deep equality between the `actual` and `expected` parameters. "Deep" equality means that the enumerable "own" properties of child objects are recursively evaluated also by the following rules.

Comparison details

- Primitive values are compared using the SameValue Comparison, used by Object.is().
- Type tags of objects should be the same.
- [[Prototype]] of objects are compared using the Strict Equality Comparison.
- Only enumerable "own" properties are considered.
- Error names and messages are always compared, even if these are not enumerable properties.
- Enumerable own Symbol properties are compared as well.

- Object wrappers are compared both as objects and unwrapped values.
- Object properties are compared unordered.
- Map keys and Set items are compared unordered.
- Recursion stops when both sides differ or both sides encounter a circular reference.
- WeakMap and WeakSet comparison does not rely on their values. See below for further details.

```
const assert = require('assert').strict;

assert.deepStrictEqual({ a: 1 }, { a: '1' });
// AssertionError: { a: 1 } deepStrictEqual { a: '1' }
// because 1 !== '1' using SameValue comparison

// The following objects don't have own properties
const date = new Date();
const object = {};
const fakeDate = {};
Object.setPrototypeOf(fakeDate, Date.prototype);

assert.deepStrictEqual(object, fakeDate);
// AssertionError: {} deepStrictEqual Date {}
// Different [[Prototype]]

assert.deepStrictEqual(date, fakeDate);
// AssertionError: 2017-03-11T14:25:31.849Z deepStrictEqual Date {}
// Different type tags

assert.deepStrictEqual(NaN, NaN);
// OK, because of the SameValue comparison

assert.deepStrictEqual(new Number(1), new Number(2));
// Fails because the wrapped number is unwrapped and compared as well.
assert.deepStrictEqual(new String('foo'), Object('foo'));
// OK because the object and the string are identical when unwrapped.

assert.deepStrictEqual(-0, -0);
// OK
assert.deepStrictEqual(0, -0);
// AssertionError: 0 deepStrictEqual -0

const symbol1 = Symbol();
const symbol2 = Symbol();
assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol1]: 1 });
// OK, because it is the same symbol on both objects.
assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol2]: 1 });
// Fails because symbol1 !== symbol2!

const weakMap1 = new WeakMap();
const weakMap2 = new WeakMap([[{}, {}]]);
const weakMap3 = new WeakMap();
weakMap3.unequal = true;

assert.deepStrictEqual(weakMap1, weakMap2);
// OK, because it is impossible to compare the entries
assert.deepStrictEqual(weakMap1, weakMap3);
// Fails because weakMap3 has a property that weakMap1 does not contain!
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the

`message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an Error then it will be thrown instead of the `AssertionError`.

assert.doesNotThrow(block[, error][, message])

▸ History

- `block` <Function>
- `error` <RegExp> | <Function>
- `message` <any>

Asserts that the function `block` does not throw an error. See assert.throws() for more details.

Please note: Using `assert.doesNotThrow()` is actually not useful because there is no benefit by catching an error and then rethrowing it. Instead, consider adding a comment next to the specific code path that should not throw and keep error messages as expressive as possible.

When `assert.doesNotThrow()` is called, it will immediately call the `block` function.

If an error is thrown and it is the same type as that specified by the `error` parameter, then an `AssertionError` is thrown. If the error is of a different type, or if the `error` parameter is undefined, the error is propagated back to the caller.

The following, for instance, will throw the TypeError because there is no matching error type in the assertion:

```
assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError
);
```

However, the following will result in an `AssertionError` with the message 'Got unwanted exception (TypeError)..':

```
assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  TypeError
);
```

If an `AssertionError` is thrown and a value is provided for the `message` parameter, the value of `message` will be appended to the `AssertionError` message:

```
assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  TypeError,
  'Whoops'
);
// Throws: AssertionError: Got unwanted exception (TypeError). Whoops
```

assert.equal(actual, expected[, message])

Added in: v0.1.21

- `actual` <any>
- `expected` <any>

- message <any>

Strict mode

An alias of assert.strictEqual() .

Legacy mode

Stability: 0 - Deprecated: Use assert.strictEqual() instead.

Tests shallow, coercive equality between the actual and expected parameters using the Abstract Equality Comparison (==).

```
const assert = require('assert');

assert.equal(1, 1);
// OK, 1 == 1
assert.equal(1, '1');
// OK, 1 == '1'

assert.equal(1, 2);
// AssertionError: 1 == 2
assert.equal({ a: { b: 1 } }, { a: { b: 1 } });
//AssertionError: { a: { b: 1 } } == { a: { b: 1 } }
```

If the values are not equal, an AssertionError is thrown with a message property set equal to the value of the message parameter. If the message parameter is undefined, a default error message is assigned. If the message parameter is an instance of an Error then it will be thrown instead of the AssertionError.

assert.fail([message])

assert.fail(actual, expected[, message[, operator[, stackStartFunction]]])

Added in: v0.1.21

- actual <any>
- expected <any>
- message <any> **Default:** 'Failed'
- operator <string> **Default:** '!='
- stackStartFunction <Function> **Default:** assert.fail

Throws an AssertionError. If message is falsy, the error message is set as the values of actual and expected separated by the provided operator. If the message parameter is an instance of an Error then it will be thrown instead of the AssertionError. If just the two actual and expected arguments are provided, operator will default to '!='. If message is provided only it will be used as the error message, the other arguments will be stored as properties on the thrown object. If stackStartFunction is provided, all stack frames above that function will be removed from stacktrace (see Error.captureStackTrace). If no arguments are given, the default message Failed will be used.

```
const assert = require('assert').strict;

assert.fail(1, 2, undefined, '>');
// AssertionError [ERR_ASSERTION]: 1 > 2

assert.fail(1, 2, 'fail');
// AssertionError [ERR_ASSERTION]: fail

assert.fail(1, 2, 'whoops', '>');
// AssertionError [ERR_ASSERTION]: whoops

assert.fail(1, 2, new TypeError('need array'));
// TypeError: need array
```

Note: In the last two cases `actual`, `expected`, and `operator` have no influence on the error message.

```
assert.fail();
// AssertionError [ERR_ASSERTION]: Failed

assert.fail('boom');
// AssertionError [ERR_ASSERTION]: boom

assert.fail('a', 'b');
// AssertionError [ERR_ASSERTION]: 'a' != 'b'
```

Example use of `stackStartFunction` for truncating the exception's stacktrace:

```
function suppressFrame() {
  assert.fail('a', 'b', undefined, '!==', suppressFrame);
}
suppressFrame();
// AssertionError [ERR_ASSERTION]: 'a' !== 'b'
//     at repl:1:1
//     at ContextifyScript.Script.runInThisContext (vm.js:44:33)
//     ...
```

assert.ifError(value)

Added in: v0.1.97

- `value` <any>

Throws `value` if `value` is truthy. This is useful when testing the `error` argument in callbacks.

```
const assert = require('assert').strict;

assert.ifError(null);
// OK
assert.ifError(0);
// OK
assert.ifError(1);
// Throws 1
assert.ifError('error');
// Throws 'error'
assert.ifError(new Error());
// Throws Error
```

assert.notDeepEqual(actual, expected[, message])

▶ History

- `actual` <any>
- `expected` <any>
- `message` <any>

Strict mode

An alias of `assert.notDeepStrictEqual()` .

Legacy mode

Stability: 0 - Deprecated: Use `assert.notDeepStrictEqual()` instead.

Tests for any deep inequality. Opposite of `assert.deepEqual()` .

```js
const assert = require('assert');

const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    b: 2
  }
};
const obj3 = {
  a: {
    b: 1
  }
};
const obj4 = Object.create(obj1);

assert.notDeepEqual(obj1, obj1);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj2);
// OK: obj1 and obj2 are not deeply equal

assert.notDeepEqual(obj1, obj3);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj4);
// OK: obj1 and obj4 are not deeply equal
```

If the values are deeply equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an Error then it will be thrown instead of the `AssertionError`.

assert.notDeepStrictEqual(actual, expected[, message])

▸ History

- `actual` <any>
- `expected` <any>
- `message` <any>

Tests for deep strict inequality. Opposite of assert.deepStrictEqual().

```js
const assert = require('assert').strict;

assert.notDeepStrictEqual({ a: 1 }, { a: '1' });
// OK
```

If the values are deeply and strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an Error then it will be thrown instead of the `AssertionError`.

assert.notEqual(actual, expected[, message])

Added in: v0.1.21

- `actual` <any>
- `expected` <any>
- `message` <any>

Strict mode

An alias of `assert.notStrictEqual()`.

Legacy mode

Stability: 0 - Deprecated: Use `assert.notStrictEqual()` instead.

Tests shallow, coercive inequality with the Abstract Equality Comparison (`!=`).

```
const assert = require('assert');

assert.notEqual(1, 2);
// OK

assert.notEqual(1, 1);
// AssertionError: 1 != 1

assert.notEqual(1, '1');
// AssertionError: 1 != '1'
```

If the values are equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.notStrictEqual(actual, expected[, message])

Added in: v0.1.21

- `actual` <any>
- `expected` <any>
- `message` <any>

Tests strict inequality between the `actual` and `expected` parameters as determined by the SameValue Comparison.

```
const assert = require('assert').strict;

assert.notStrictEqual(1, 2);
// OK

assert.notStrictEqual(1, 1);
// AssertionError: 1 !== 1

assert.notStrictEqual(1, '1');
// OK
```

If the values are strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.ok(value[, message])

Added in: v0.1.21

- `value` <any>
- `message` <any>

Tests if `value` is truthy. It is equivalent to `assert.equal(!!value, true, message)`.

If `value` is not truthy, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is `undefined`, a default error message is assigned. If the `message` parameter is an instance of an Error then it will be thrown instead of the `AssertionError`.

```
const assert = require('assert').strict;

assert.ok(true);
// OK
assert.ok(1);
// OK
assert.ok(false);
// throws "AssertionError: false == true"
assert.ok(0);
// throws "AssertionError: 0 == true"
assert.ok(false, 'it\'s false');
// throws "AssertionError: it's false"
```

assert.strictEqual(actual, expected[, message])

Added in: v0.1.21

- `actual` <any>
- `expected` <any>
- `message` <any>

Tests strict equality between the `actual` and `expected` parameters as determined by the SameValue Comparison.

```
const assert = require('assert').strict;

assert.strictEqual(1, 2);
// AssertionError: 1 === 2

assert.strictEqual(1, 1);
// OK

assert.strictEqual(1, '1');
// AssertionError: 1 === '1'
```

If the values are not strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an Error then it will be thrown instead of the `AssertionError`.

assert.throws(block[, error][, message])

▸ History

- `block` <Function>
- `error` <RegExp> | <Function> | <object>
- `message` <any>

Expects the function `block` to throw an error.

If specified, `error` can be a constructor, `RegExp` , a validation function, or an object where each property will be tested for.

If specified, `message` will be the message provided by the `AssertionError` if the block fails to throw.

Validate instanceof using constructor:

```
assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  Error
);
```

Validate error message using `RegExp` :

Using a regular expression runs `.toString` on the error object, and will therefore also include the error name.

```
assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  /^Error: Wrong value$/
);
```

Custom error validation:

```
assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  function(err) {
    if ((err instanceof Error) && /value/.test(err)) {
      return true;
    }
  },
  'unexpected error'
);
```

Custom error object / error instance:

```
assert.throws(
  () => {
    const err = new TypeError('Wrong value');
    err.code = 404;
    throw err;
  },
  {
    name: 'TypeError',
    message: 'Wrong value'
    // Note that only properties on the error object will be tested!
  }
);
```

Note that `error` can not be a string. If a string is provided as the second argument, then `error` is assumed to be omitted and the string will be used for `message` instead. This can lead to easy-to-miss mistakes. Please read the example below carefully if using a string as the second argument gets considered:

```
function throwingFirst() {
  throw new Error('First');
}
function throwingSecond() {
  throw new Error('Second');
}
function notThrowing() {}

// The second argument is a string and the input function threw an Error.
// In that case both cases do not throw as neither is going to try to
// match for the error message thrown by the input function!
assert.throws(throwingFirst, 'Second');
assert.throws(throwingSecond, 'Second');

// The string is only used (as message) in case the function does not throw:
assert.throws(notThrowing, 'Second');
// AssertionError [ERR_ASSERTION]: Missing expected exception: Second

// If it was intended to match for the error message do this instead:
assert.throws(throwingSecond, /Second$/);
// Does not throw because the error messages match.
assert.throws(throwingFirst, /Second$/);
// Throws a error:
// Error: First
//     at throwingFirst (repl:2:9)
```

Due to the confusing notation, it is recommended not to use a string as the second argument. This might lead to difficult-to-spot errors.

Caveats

For the following cases, consider using ES2015 `Object.is()`, which uses the SameValueZero comparison.

```
const a = 0;
const b = -a;
assert.notStrictEqual(a, b);
// AssertionError: 0 !== -0
// Strict Equality Comparison doesn't distinguish between -0 and +0...
assert(!Object.is(a, b));
// but Object.is() does!

const str1 = 'foo';
const str2 = 'foo';
assert.strictEqual(str1 / 1, str2 / 1);
// AssertionError: NaN === NaN
// Strict Equality Comparison can't be used to check NaN...
assert(Object.is(str1 / 1, str2 / 1));
// but Object.is() can!
```

For more information, see MDN's guide on equality comparisons and sameness.

Async Hooks

Stability: 1 - Experimental

The `async_hooks` module provides an API to register callbacks tracking the lifetime of asynchronous resources created inside a Node.js application. It can be accessed using:

```
const async_hooks = require('async_hooks');
```

Terminology

An asynchronous resource represents an object with an associated callback. This callback may be called multiple times, for example, the `connection` event in `net.createServer`, or just a single time like in `fs.open`. A resource can also be closed before the callback is called. AsyncHook does not explicitly distinguish between these different cases but will represent them as the abstract concept that is a resource.

Public API

Overview

Following is a simple overview of the public API.

```js
const async_hooks = require('async_hooks');

// Return the ID of the current execution context.
const eid = async_hooks.executionAsyncId();

// Return the ID of the handle responsible for triggering the callback of the
// current execution scope to call.
const tid = async_hooks.triggerAsyncId();

// Create a new AsyncHook instance. All of these callbacks are optional.
const asyncHook =
  async_hooks.createHook({ init, before, after, destroy, promiseResolve });

// Allow callbacks of this AsyncHook instance to call. This is not an implicit
// action after running the constructor, and must be explicitly run to begin
// executing callbacks.
asyncHook.enable();

// Disable listening for new asynchronous events.
asyncHook.disable();

//
// The following are the callbacks that can be passed to createHook().
//

// init is called during object construction. The resource may not have
// completed construction when this callback runs, therefore all fields of the
// resource referenced by "asyncId" may not have been populated.
function init(asyncId, type, triggerAsyncId, resource) { }

// before is called just before the resource's callback is called. It can be
// called 0-N times for handles (e.g. TCPWrap), and will be called exactly 1
// time for requests (e.g. FSReqWrap).
function before(asyncId) { }

// after is called just after the resource's callback has finished.
function after(asyncId) { }

// destroy is called when an AsyncWrap instance is destroyed.
function destroy(asyncId) { }

// promiseResolve is called only for promise resources, when the
// `resolve` function passed to the `Promise` constructor is invoked
// (either directly or through other means of resolving a promise).
function promiseResolve(asyncId) { }
```

async_hooks.createHook(callbacks)

Added in: v8.1.0

- `callbacks` <Object> The Hook Callbacks to register
  - `init` <Function> The init callback .
  - `before` <Function> The before callback .
  - `after` <Function> The after callback .
  - `destroy` <Function> The destroy callback .
- Returns: <AsyncHook> Instance used for disabling and enabling hooks

Registers functions to be called for different lifetime events of each async operation.

The callbacks `init()` / `before()` / `after()` / `destroy()` are called for the respective asynchronous event during a resource's lifetime.

All callbacks are optional. For example, if only resource cleanup needs to be tracked, then only the `destroy` callback needs to be passed. The specifics of all functions that can be passed to `callbacks` is in the Hook Callbacks section.

```
const async_hooks = require('async_hooks');

const asyncHook = async_hooks.createHook({
  init(asyncId, type, triggerAsyncId, resource) { },
  destroy(asyncId) { }
});
```

Note that the callbacks will be inherited via the prototype chain:

```
class MyAsyncCallbacks {
  init(asyncId, type, triggerAsyncId, resource) { }
  destroy(asyncId) {}
}

class MyAddedCallbacks extends MyAsyncCallbacks {
  before(asyncId) { }
  after(asyncId) { }
}

const asyncHook = async_hooks.createHook(new MyAddedCallbacks());
```

Error Handling

If any `AsyncHook` callbacks throw, the application will print the stack trace and exit. The exit path does follow that of an uncaught exception, but all `uncaughtException` listeners are removed, thus forcing the process to exit. The `'exit'` callbacks will still be called unless the application is run with `--abort-on-uncaught-exception`, in which case a stack trace will be printed and the application exits, leaving a core file.

The reason for this error handling behavior is that these callbacks are running at potentially volatile points in an object's lifetime, for example during class construction and destruction. Because of this, it is deemed necessary to bring down the process quickly in order to prevent an unintentional abort in the future. This is subject to change in the future if a comprehensive analysis is performed to ensure an exception can follow the normal control flow without unintentional side effects.

Printing in AsyncHooks callbacks

Because printing to the console is an asynchronous operation, `console.log()` will cause the AsyncHooks callbacks to be called. Using `console.log()` or similar asynchronous operations inside an AsyncHooks callback function will thus cause an infinite recursion. An easy solution to this when debugging is to use a synchronous logging operation such as `fs.writeSync(1, msg)`. This will print to stdout because `1` is the file descriptor for stdout and will not invoke AsyncHooks recursively because it is synchronous.

```
const fs = require('fs');
const util = require('util');

function debug(...args) {
  // use a function like this one when debugging inside an AsyncHooks callback
  fs.writeSync(1, `${util.format(...args)}\n`);
}
```

If an asynchronous operation is needed for logging, it is possible to keep track of what caused the asynchronous operation using the information provided by AsyncHooks itself. The logging should then be skipped when it was the logging itself that caused AsyncHooks callback to call. By doing this the otherwise infinite recursion is broken.

asyncHook.enable()

- Returns: <AsyncHook> A reference to `asyncHook`.

Enable the callbacks for a given `AsyncHook` instance. If no callbacks are provided enabling is a noop.

The `AsyncHook` instance is disabled by default. If the `AsyncHook` instance should be enabled immediately after creation, the following pattern can be used.

```
const async_hooks = require('async_hooks');

const hook = async_hooks.createHook(callbacks).enable();
```

asyncHook.disable()

- Returns: <AsyncHook> A reference to `asyncHook`.

Disable the callbacks for a given `AsyncHook` instance from the global pool of AsyncHook callbacks to be executed. Once a hook has been disabled it will not be called again until enabled.

For API consistency `disable()` also returns the `AsyncHook` instance.

Hook Callbacks

Key events in the lifetime of asynchronous events have been categorized into four areas: instantiation, before/after the callback is called, and when the instance is destroyed.

init(asyncId, type, triggerAsyncId, resource)

- `asyncId` <number> A unique ID for the async resource.
- `type` <string> The type of the async resource.
- `triggerAsyncId` <number> The unique ID of the async resource in whose execution context this async resource was created.
- `resource` <Object> Reference to the resource representing the async operation, needs to be released during *destroy*.

Called when a class is constructed that has the *possibility* to emit an asynchronous event. This *does not* mean the instance must call `before` / `after` before `destroy` is called, only that the possibility exists.

This behavior can be observed by doing something like opening a resource then closing it before the resource can be used. The following snippet demonstrates this.

```
require('net').createServer().listen(function() { this.close(); });
// OR
clearTimeout(setTimeout(() => {}, 10));
```

Every new resource is assigned an ID that is unique within the scope of the current process.

type

The `type` is a string identifying the type of resource that caused `init` to be called. Generally, it will correspond to the name of the resource's constructor.

```
FSEVENTWRAP, FSREQWRAP, GETADDRINFOREQWRAP, GETNAMEINFOREQWRAP, HTTPPARSER,
JSSTREAM, PIPECONNECTWRAP, PIPEWRAP, PROCESSWRAP, QUERYWRAP, SHUTDOWNWRAP,
SIGNALWRAP, STATWATCHER, TCPCONNECTWRAP, TCPSERVER, TCPWRAP, TIMERWRAP, TTYWRAP,
UDPSENDWRAP, UDPWRAP, WRITEWRAP, ZLIB, SSLCONNECTION, PBKDF2REQUEST,
RANDOMBYTESREQUEST, TLSWRAP, Timeout, Immediate, TickObject
```

There is also the `PROMISE` resource type, which is used to track `Promise` instances and asynchronous work scheduled by them.

Users are able to define their own `type` when using the public embedder API.

Note: It is possible to have type name collisions. Embedders are encouraged to use unique prefixes, such as the npm package name, to prevent collisions when listening to the hooks.

`triggerId`

`triggerAsyncId` is the `asyncId` of the resource that caused (or "triggered") the new resource to initialize and that caused `init` to call. This is different from `async_hooks.executionAsyncId()` that only shows *when* a resource was created, while `triggerAsyncId` shows *why* a resource was created.

The following is a simple demonstration of `triggerAsyncId`:

```
async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = async_hooks.executionAsyncId();
    fs.writeSync(
      1, `${type}(${asyncId}): trigger: ${triggerAsyncId} execution: ${eid}\n`);
  }
}).enable();

require('net').createServer((conn) => {}).listen(8080);
```

Output when hitting the server with `nc localhost 8080`:

```
TCPSERVERWRAP(2): trigger: 1 execution: 1
TCPWRAP(4): trigger: 2 execution: 0
```

The `TCPSERVERWRAP` is the server which receives the connections.

The `TCPWRAP` is the new connection from the client. When a new connection is made the `TCPWrap` instance is immediately constructed. This happens outside of any JavaScript stack (side note: a `executionAsyncId()` of `0` means it's being executed from C++, with no JavaScript stack above it). With only that information, it would be impossible to link resources together in terms of what caused them to be created, so `triggerAsyncId` is given the task of propagating what resource is responsible for the new resource's existence.

`resource`

`resource` is an object that represents the actual async resource that has been initialized. This can contain useful information that can vary based on the value of `type`. For instance, for the `GETADDRINFOREQWRAP` resource type, `resource` provides the hostname used when looking up the IP address for the hostname in `net.Server.listen()`. The API for accessing this information is currently not considered public, but using the Embedder API, users can provide and document their own resource objects. For example, such a resource object could contain the SQL query being executed.

In the case of Promises, the `resource` object will have `promise` property that refers to the Promise that is being initialized, and a `isChainedPromise` property, set to `true` if the promise has a parent promise, and `false` otherwise. For example, in the case of `b = a.then(handler)`, `a` is considered a parent Promise of `b`. Here, `b` is considered a chained promise.

In some cases the resource object is reused for performance reasons, it is thus not safe to use it as a key in a `WeakMap` or add properties to it.

Asynchronous context example

The following is an example with additional information about the calls to `init` between the `before` and `after` calls, specifically what the callback to `listen()` will look like. The output formatting is slightly more elaborate to make calling context easier to see.

```
let indent = 0;
async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = async_hooks.executionAsyncId();
    const indentStr = ' '.repeat(indent);
    fs.writeSync(
      1,
      `${indentStr}${type}(${asyncId}):` +
      ` trigger: ${triggerAsyncId} execution: ${eid}\n`);
  },
  before(asyncId) {
    const indentStr = ' '.repeat(indent);
    fs.writeSync(1, `${indentStr}before:  ${asyncId}\n`);
    indent += 2;
  },
  after(asyncId) {
    indent -= 2;
    const indentStr = ' '.repeat(indent);
    fs.writeSync(1, `${indentStr}after:   ${asyncId}\n`);
  },
  destroy(asyncId) {
    const indentStr = ' '.repeat(indent);
    fs.writeSync(1, `${indentStr}destroy: ${asyncId}\n`);
  },
}).enable();

require('net').createServer(() => {}).listen(8080, () => {
  // Let's wait 10ms before logging the server started.
  setTimeout(() => {
    console.log('>>>', async_hooks.executionAsyncId());
  }, 10);
});
```

Output from only starting the server:

```
TCPSERVERWRAP(2): trigger: 1 execution: 1
TickObject(3): trigger: 2 execution: 1
before:  3
  Timeout(4): trigger: 3 execution: 3
  TIMERWRAP(5): trigger: 3 execution: 3
after:   3
destroy: 3
before:  5
  before:  4
    TTYWRAP(6): trigger: 4 execution: 4
    SIGNALWRAP(7): trigger: 4 execution: 4
    TTYWRAP(8): trigger: 4 execution: 4
>>> 4
    TickObject(9): trigger: 4 execution: 4
  after:   4
after:   5
before:  9
after:   9
destroy: 4
destroy: 9
destroy: 5
```

As illustrated in the example, `executionAsyncId()` and `execution` each specify the value of the current execution context; which is delineated by calls to `before` and `after`.

Only using `execution` to graph resource allocation results in the following:

```
TTYWRAP(6) -> Timeout(4) -> TIMERWRAP(5) -> TickObject(3) -> root(1)
```

The `TCPSERVERWRAP` is not part of this graph, even though it was the reason for `console.log()` being called. This is because binding to a port without a hostname is a *synchronous* operation, but to maintain a completely asynchronous API the user's callback is placed in a `process.nextTick()`.

The graph only shows *when* a resource was created, not *why*, so to track the *why* use `triggerAsyncId`.

before(asyncId)

- `asyncId` <number>

When an asynchronous operation is initiated (such as a TCP server receiving a new connection) or completes (such as writing data to disk) a callback is called to notify the user. The `before` callback is called just before said callback is executed. `asyncId` is the unique identifier assigned to the resource about to execute the callback.

The `before` callback will be called 0 to N times. The `before` callback will typically be called 0 times if the asynchronous operation was cancelled or, for example, if no connections are received by a TCP server. Persistent asynchronous resources like a TCP server will typically call the `before` callback multiple times, while other operations like `fs.open()` will call it only once.

after(asyncId)

- `asyncId` <number>

Called immediately after the callback specified in `before` is completed.

Note: If an uncaught exception occurs during execution of the callback, then `after` will run *after* the `'uncaughtException'` event is emitted or a `domain`'s handler runs.

destroy(asyncId)

- `asyncId` <number>

Called after the resource corresponding to `asyncId` is destroyed. It is also called asynchronously from the embedder API `emitDestroy()`.

Note: Some resources depend on garbage collection for cleanup, so if a reference is made to the `resource` object passed to `init` it is possible that `destroy` will never be called, causing a memory leak in the application. If the resource does not depend on garbage collection, then this will not be an issue.

promiseResolve(asyncId)

- `asyncId` <number>

Called when the `resolve` function passed to the `Promise` constructor is invoked (either directly or through other means of resolving a promise).

Note that `resolve()` does not do any observable synchronous work.

Note: This does not necessarily mean that the `Promise` is fulfilled or rejected at this point, if the `Promise` was resolved by assuming the state of another `Promise`.

```
new Promise((resolve) => resolve(true)).then((a) => {});
```

calls the following callbacks:

```
init for PROMISE with id 5, trigger id: 1
  promise resolve 5      # corresponds to resolve(true)
init for PROMISE with id 6, trigger id: 5  # the Promise returned by then()
  before 6               # the then() callback is entered
  promise resolve 6      # the then() callback resolves the promise by returning
  after 6
```

async_hooks.executionAsyncId()

▸ History

- Returns: <number> The `asyncId` of the current execution context. Useful to track when something calls.

```
const async_hooks = require('async_hooks');

console.log(async_hooks.executionAsyncId());  // 1 - bootstrap
fs.open(path, 'r', (err, fd) => {
  console.log(async_hooks.executionAsyncId());  // 6 - open()
});
```

The ID returned from `executionAsyncId()` is related to execution timing, not causality (which is covered by `triggerAsyncId()`):

```
const server = net.createServer(function onConnection(conn) {
  // Returns the ID of the server, not of the new connection, because the
  // onConnection callback runs in the execution scope of the server's
  // MakeCallback().
  async_hooks.executionAsyncId();

}).listen(port, function onListening() {
  // Returns the ID of a TickObject (i.e. process.nextTick()) because all
  // callbacks passed to .listen() are wrapped in a nextTick().
  async_hooks.executionAsyncId();
});
```

Note that promise contexts may not get precise executionAsyncIds by default. See the section on promise execution tracking .

async_hooks.triggerAsyncId()

- Returns: <number> The ID of the resource responsible for calling the callback that is currently being executed.

```
const server = net.createServer((conn) => {
  // The resource that caused (or triggered) this callback to be called
  // was that of the new connection. Thus the return value of triggerAsyncId()
  // is the asyncId of "conn".
  async_hooks.triggerAsyncId();

}).listen(port, () => {
  // Even though all callbacks passed to .listen() are wrapped in a nextTick()
  // the callback itself exists because the call to the server's .listen()
  // was made. So the return value would be the ID of the server.
  async_hooks.triggerAsyncId();
});
```

Note that promise contexts may not get valid triggerAsyncIds by default. See the section on promise execution tracking .

Promise execution tracking

By default, promise executions are not assigned asyncIds due to the relatively expensive nature of the promise introspection API provided by V8. This means that programs using promises or `async` / `await` will not get correct execution and trigger ids for promise callback contexts by default.

Here's an example:

```
const ah = require('async_hooks');
Promise.resolve(1729).then(() => {
  console.log(`eid ${ah.executionAsyncId()} tid ${ah.triggerAsyncId()}`);
});
// produces:
// eid 1 tid 0
```

Observe that the `then` callback claims to have executed in the context of the outer scope even though there was an asynchronous hop involved. Also note that the triggerAsyncId value is 0, which means that we are missing context about the resource that caused (triggered) the `then` callback to be executed.

Installing async hooks via `async_hooks.createHook` enables promise execution tracking. Example:

```
const ah = require('async_hooks');
ah.createHook({ init() {} }).enable(); // forces PromiseHooks to be enabled.
Promise.resolve(1729).then(() => {
  console.log(`eid ${ah.executionAsyncId()} tid ${ah.triggerAsyncId()}`);
});
// produces:
// eid 7 tid 6
```

In this example, adding any actual hook function enabled the tracking of promises. There are two promises in the example above; the promise created by `Promise.resolve()` and the promise returned by the call to `then`. In the example above, the first promise got the asyncId 6 and the latter got asyncId 7. During the execution of the `then` callback, we are executing in the context of promise with asyncId 7. This promise was triggered by async resource 6.

Another subtlety with promises is that `before` and `after` callbacks are run only on chained promises. That means promises not created by `then` / `catch` will not have the `before` and `after` callbacks fired on them. For more details see the details of the V8 PromiseHooks API.

JavaScript Embedder API

Library developers that handle their own asynchronous resources performing tasks like I/O, connection pooling, or managing callback queues may use the `AsyncWrap` JavaScript API so that all the appropriate callbacks are called.

class AsyncResource()

The class `AsyncResource` is designed to be extended by the embedder's async resources. Using this, users can easily trigger the lifetime events of their own resources.

The `init` hook will trigger when an `AsyncResource` is instantiated.

The following is an overview of the `AsyncResource` API.

```js
const { AsyncResource, executionAsyncId } = require('async_hooks');

// AsyncResource() is meant to be extended. Instantiating a
// new AsyncResource() also triggers init. If triggerAsyncId is omitted then
// async_hook.executionAsyncId() is used.
const asyncResource = new AsyncResource(
  type, { triggerAsyncId: executionAsyncId(), requireManualDestroy: false }
);

// Run a function in the execution context of the resource. This will
// * establish the context of the resource
// * trigger the AsyncHooks before callbacks
// * call the provided function `fn` with the supplied arguments
// * trigger the AsyncHooks after callbacks
// * restore the original execution context
asyncResource.runInAsyncScope(fn, thisArg, ...args);

// Call AsyncHooks destroy callbacks.
asyncResource.emitDestroy();

// Return the unique ID assigned to the AsyncResource instance.
asyncResource.asyncId();

// Return the trigger ID for the AsyncResource instance.
asyncResource.triggerAsyncId();

// Call AsyncHooks before callbacks.
// Deprecated: Use asyncResource.runInAsyncScope instead.
asyncResource.emitBefore();

// Call AsyncHooks after callbacks.
// Deprecated: Use asyncResource.runInAsyncScope instead.
asyncResource.emitAfter();
```

AsyncResource(type[, options])

- `type` <string> The type of async event.
- `options` <Object>
  - `triggerAsyncId` <number> The ID of the execution context that created this async event. **Default:** `executionAsyncId()`.
  - `requireManualDestroy` <boolean> Disables automatic `emitDestroy` when the object is garbage collected. This usually does not need to be set (even if `emitDestroy` is called manually), unless the resource's asyncId is retrieved and the sensitive API's `emitDestroy` is called with it. **Default:** `false`

Example usage:

```js
class DBQuery extends AsyncResource {
  constructor(db) {
    super('DBQuery');
    this.db = db;
  }

  getInfo(query, callback) {
    this.db.get(query, (err, data) => {
      this.runInAsyncScope(callback, null, err, data);
    });
  }

  close() {
    this.db = null;
    this.emitDestroy();
  }
}
```

asyncResource.runInAsyncScope(fn[, thisArg, ...args])

Added in: v9.6.0

- `fn` <Function> The function to call in the execution context of this async resource.
- `thisArg` <any> The receiver to be used for the function call.
- `...args` <any> Optional arguments to pass to the function.

Call the provided function with the provided arguments in the execution context of the async resource. This will establish the context, trigger the AsyncHooks before callbacks, call the function, trigger the AsyncHooks after callbacks, and then restore the original execution context.

asyncResource.emitBefore()

Deprecated since: v9.6.0

Stability: 0 - Deprecated: Use asyncResource.runInAsyncScope() instead.

Call all `before` callbacks to notify that a new asynchronous execution context is being entered. If nested calls to `emitBefore()` are made, the stack of `asyncId` s will be tracked and properly unwound.

`before` and `after` calls must be unwound in the same order that they are called. Otherwise, an unrecoverable exception will occur and the process will abort. For this reason, the `emitBefore` and `emitAfter` APIs are considered deprecated. Please use `runInAsyncScope`, as it provides a much safer alternative.

asyncResource.emitAfter()

Deprecated since: v9.6.0

Stability: 0 - Deprecated: Use asyncResource.runInAsyncScope() instead.

Call all `after` callbacks. If nested calls to `emitBefore()` were made, then make sure the stack is unwound properly. Otherwise an error will be thrown.

If the user's callback throws an exception, `emitAfter()` will automatically be called for all `asyncId` s on the stack if the error is handled by a domain or `'uncaughtException'` handler.

`before` and `after` calls must be unwound in the same order that they are called. Otherwise, an unrecoverable exception will occur and the process will abort. For this reason, the `emitBefore` and `emitAfter` APIs are considered deprecated. Please use `runInAsyncScope`, as it provides a much safer alternative.

asyncResource.emitDestroy()

Call all `destroy` hooks. This should only ever be called once. An error will be thrown if it is called more than once. This**must** be manually called. If the resource is left to be collected by the GC then the `destroy` hooks will never be called.

asyncResource.asyncId()

- Returns: <number> The unique `asyncId` assigned to the resource.

asyncResource.triggerAsyncId()

- Returns: <number> The same `triggerAsyncId` that is passed to the `AsyncResource` constructor.

Buffer

Stability: 2 - Stable

Prior to the introduction of TypedArray, the JavaScript language had no mechanism for reading or manipulating streams of binary data. The `Buffer` class was introduced as part of the Node.js API to make it possible to interact with octet streams in the context of things like TCP streams and file system operations.

With TypedArray now available, the `Buffer` class implements the Uint8Array API in a manner that is more optimized and suitable for Node.js.

Instances of the `Buffer` class are similar to arrays of integers but correspond to fixed-sized, raw memory allocations outside the V8 heap. The size of the `Buffer` is established when it is created and cannot be changed.

The `Buffer` class is within the global scope, making it unlikely that one would need to ever use `require('buffer').Buffer`.

```
// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);

// Creates a Buffer of length 10, filled with 0x1.
const buf2 = Buffer.alloc(10, 1);

// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using either fill() or write().
const buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing [0x1, 0x2, 0x3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing UTF-8 bytes [0x74, 0xc3, 0xa9, 0x73, 0x74].
const buf5 = Buffer.from('tést');

// Creates a Buffer containing Latin-1 bytes [0x74, 0xe9, 0x73, 0x74].
const buf6 = Buffer.from('tést', 'latin1');
```

`Buffer.from()`, `Buffer.alloc()`, and `Buffer.allocUnsafe()`

In versions of Node.js prior to 6.0.0, `Buffer` instances were created using the `Buffer` constructor function, which allocates the returned `Buffer` differently based on what arguments are provided:

- Passing a number as the first argument to `Buffer()` (e.g. `new Buffer(10)`) allocates a new `Buffer` object of the specified size. Prior to Node.js 8.0.0, the memory allocated for such `Buffer` instances is *not* initialized and *can contain sensitive data*. Such `Buffer` instances *must* be subsequently initialized by using either `buf.fill(0)` or by writing to the entire `Buffer`. While this behavior is *intentional* to improve

performance, development experience has demonstrated that a more explicit distinction is required between creating a fast-but-uninitialized `Buffer` versus creating a slower-but-safer `Buffer`. Starting in Node.js 8.0.0, `Buffer(num)` and `new Buffer(num)` will return a `Buffer` with initialized memory.
- Passing a string, array, or `Buffer` as the first argument copies the passed object's data into the `Buffer`.
- Passing an `ArrayBuffer` or a `SharedArrayBuffer` returns a `Buffer` that shares allocated memory with the given array buffer.

Because the behavior of `new Buffer()` is different depending on the type of the first argument, security and reliability issues can be inadvertently introduced into applications when argument validation or `Buffer` initialization is not performed.

To make the creation of `Buffer` instances more reliable and less error-prone, the various forms of the `new Buffer()` constructor have been **deprecated** and replaced by separate `Buffer.from()`, `Buffer.alloc()`, and `Buffer.allocUnsafe()` methods.

Developers should migrate all existing uses of the `new Buffer()` *constructors to one of these new APIs.*

- `Buffer.from(array)` returns a new `Buffer` that *contains a copy* of the provided octets.
- `Buffer.from(arrayBuffer[, byteOffset [, length]])` returns a new `Buffer` that *shares the same allocated memory* as the given `ArrayBuffer`.
- `Buffer.from(buffer)` returns a new `Buffer` that *contains a copy* of the contents of the given `Buffer`.
- `Buffer.from(string[, encoding])` returns a new `Buffer` that *contains a copy* of the provided string.
- `Buffer.alloc(size[, fill[, encoding]])` returns a new initialized `Buffer` of the specified size. This method is slower than `Buffer.allocUnsafe(size)` but guarantees that newly created `Buffer` instances never contain old data that is potentially sensitive.
- `Buffer.allocUnsafe(size)` and `Buffer.allocUnsafeSlow(size)` each return a new uninitialized `Buffer` of the specified `size`. Because the `Buffer` is uninitialized, the allocated segment of memory might contain old data that is potentially sensitive.

`Buffer` instances returned by `Buffer.allocUnsafe()` *may* be allocated off a shared internal memory pool if `size` is less than or equal to half `Buffer.poolSize`. Instances returned by `Buffer.allocUnsafeSlow()` *never* use the shared internal memory pool.

The `--zero-fill-buffers` command line option

Added in: v5.10.0

Node.js can be started using the `--zero-fill-buffers` command line option to cause all newly allocated `Buffer` instances to be zero-filled upon creation by default, including buffers returned by `new Buffer(size)`, `Buffer.allocUnsafe()`, `Buffer.allocUnsafeSlow()`, and `new SlowBuffer(size)`. Use of this flag can have a significant negative impact on performance. Use of the `--zero-fill-buffers` option is recommended only when necessary to enforce that newly allocated `Buffer` instances cannot contain old data that is potentially sensitive.

```
$ node --zero-fill-buffers
> Buffer.allocUnsafe(5);
<Buffer 00 00 00 00 00>
```

What makes `Buffer.allocUnsafe()` and `Buffer.allocUnsafeSlow()` "unsafe"?

When calling `Buffer.allocUnsafe()` and `Buffer.allocUnsafeSlow()`, the segment of allocated memory is *uninitialized* (it is not zeroed-out). While this design makes the allocation of memory quite fast, the allocated segment of memory might contain old data that is potentially sensitive. Using a `Buffer` created by `Buffer.allocUnsafe()` without *completely* overwriting the memory can allow this old data to be leaked when the `Buffer` memory is read.

While there are clear performance advantages to using `Buffer.allocUnsafe()`, extra care *must* be taken in order to avoid introducing security vulnerabilities into an application.

Buffers and Character Encodings

▸ History

When string data is stored in or extracted out of a `Buffer` instance, a character encoding may be specified.

```
const buf = Buffer.from('hello world', 'ascii');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'ascii'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>
```

The character encodings currently supported by Node.js include:

- `'ascii'` - For 7-bit ASCII data only. This encoding is fast and will strip the high bit if set.
- `'utf8'` - Multibyte encoded Unicode characters. Many web pages and other document formats use UTF-8.
- `'utf16le'` - 2 or 4 bytes, little-endian encoded Unicode characters. Surrogate pairs (U+10000 to U+10FFFF) are supported.
- `'ucs2'` - Alias of `'utf16le'`.
- `'base64'` - Base64 encoding. When creating a `Buffer` from a string, this encoding will also correctly accept "URL and Filename Safe Alphabet" as specified in RFC4648, Section 5 .
- `'latin1'` - A way of encoding the `Buffer` into a one-byte encoded string (as defined by the IANA in RFC1345 , page 63, to be the Latin-1 supplement block and C0/C1 control codes).
- `'binary'` - Alias for `'latin1'`.
- `'hex'` - Encode each byte as two hexadecimal characters.

Modern Web browsers follow the WHATWG Encoding Standard which aliases both `'latin1'` and `'ISO-8859-1'` to `'win-1252'`. This means that while doing something like `http.get()`, if the returned charset is one of those listed in the WHATWG specification it is possible that the server actually returned win-1252-encoded data, and using `'latin1'` encoding may incorrectly decode the characters.

Buffers and TypedArray

▶ History

`Buffer` instances are also `Uint8Array` instances. However, there are subtle incompatibilities with `TypedArray`. For example, while `ArrayBuffer#slice()` creates a copy of the slice, the implementation of `Buffer#slice()` creates a view over the existing `Buffer` without copying, making `Buffer#slice()` far more efficient.

It is also possible to create new `TypedArray` instances from a `Buffer` with the following caveats:

1. The `Buffer` object's memory is copied to the `TypedArray`, not shared.
2. The `Buffer` object's memory is interpreted as an array of distinct elements, and not as a byte array of the target type. That is, `new Uint32Array(Buffer.from([1, 2, 3, 4]))` creates a 4-element `Uint32Array` with elements `[1, 2, 3, 4]`, not a `Uint32Array` with a single element `[0x1020304]` or `[0x4030201]`.

It is possible to create a new `Buffer` that shares the same allocated memory as a `TypedArray` instance by using the TypeArray object's `.buffer` property.

```
const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Copies the contents of `arr`
const buf1 = Buffer.from(arr);
// Shares memory with `arr`
const buf2 = Buffer.from(arr.buffer);

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>
```

Note that when creating a `Buffer` using a TypedArray's `.buffer`, it is possible to use only a portion of the underlying ArrayBuffer by passing in `byteOffset` and `length` parameters.

```
const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);

console.log(buf.length);
// Prints: 16
```

The `Buffer.from()` and TypedArray.from() have different signatures and implementations. Specifically, the TypedArray variants accept a second argument that is a mapping function that is invoked on every element of the typed array:

- `TypedArray.from(source[, mapFn[, thisArg]])`

The `Buffer.from()` method, however, does not support the use of a mapping function:

- Buffer.from(array)
- Buffer.from(buffer)
- Buffer.from(arrayBuffer[, byteOffset [, length]])
- Buffer.from(string[, encoding])

Buffers and iteration

`Buffer` instances can be iterated over using `for..of` syntax:

```
const buf = Buffer.from([1, 2, 3]);

// Prints:
//   1
//   2
//   3
for (const b of buf) {
  console.log(b);
}
```

Additionally, the `buf.values()`, `buf.keys()`, and `buf.entries()` methods can be used to create iterators.

Class: Buffer

The `Buffer` class is a global type for dealing with binary data directly. It can be constructed in a variety of ways.

new Buffer(array)

▸ History

Stability: 0 - Deprecated: Use Buffer.from(array) instead.

- `array` <integer[]> An array of bytes to copy from.

Allocates a new `Buffer` using an `array` of octets.

```
// Creates a new Buffer containing the UTF-8 bytes of the string 'buffer'
const buf = new Buffer([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);
```

new Buffer(arrayBuffer[, byteOffset [, length]])

▸ History

Stability: 0 - Deprecated: Use Buffer.from(arrayBuffer[, byteOffset [, length]]) instead.

- `arrayBuffer` <ArrayBuffer> | <SharedArrayBuffer> An ArrayBuffer, SharedArrayBuffer or the `.buffer` property of a TypedArray.
- `byteOffset` <integer> Index of first byte to expose. **Default:** `0`
- `length` <integer> Number of bytes to expose. **Default:** `arrayBuffer.length - byteOffset`

This creates a view of the ArrayBuffer or SharedArrayBuffer without copying the underlying memory. For example, when passed a reference to the `.buffer` property of a TypedArray instance, the newly created `Buffer` will share the same allocated memory as the TypedArray.

The optional `byteOffset` and `length` arguments specify a memory range within the `arrayBuffer` that will be shared by the `Buffer`.

```
const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Shares memory with `arr`
const buf = new Buffer(arr.buffer);

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// Changing the original Uint16Array changes the Buffer also
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

new Buffer(buffer)

▸ History

Stability: 0 - Deprecated: Use Buffer.from(buffer) instead.

- `buffer` <Buffer> An existing `Buffer` to copy data from.

Copies the passed `buffer` data onto a new `Buffer` instance.

```
const buf1 = new Buffer('buffer');
const buf2 = new Buffer(buf1);

buf1[0] = 0x61;

console.log(buf1.toString());
// Prints: auffer
console.log(buf2.toString());
// Prints: buffer
```

new Buffer(size)

▸ History

Stability: 0 - Deprecated: Use Buffer.alloc() instead (also see Buffer.allocUnsafe()).

- `size` <integer> The desired length of the new `Buffer`.

Allocates a new `Buffer` of `size` bytes. If the `size` is larger than buffer.constants.MAX_LENGTH or smaller than 0, a RangeError will be thrown. A zero-length `Buffer` will be created if `size` is 0.

Prior to Node.js 8.0.0, the underlying memory for `Buffer` instances created in this way is *not initialized*. The contents of a newly created `Buffer` are unknown and *may contain sensitive data*. Use Buffer.alloc(size) instead to initialize a `Buffer` with zeroes.

```
const buf = new Buffer(10);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00 00 00 00 00 00>
```

new Buffer(string[, encoding])

▸ History

Stability: 0 - Deprecated: Use Buffer.from(string[, encoding]) instead.

- `string` <string> String to encode.
- `encoding` <string> The encoding of `string`. **Default:** `'utf8'`

Creates a new `Buffer` containing `string`. The `encoding` parameter identifies the character encoding of `string`.

```
const buf1 = new Buffer('this is a tést');
const buf2 = new Buffer('7468697320697320612074c3a97374', 'hex');

console.log(buf1.toString());
// Prints: this is a tést
console.log(buf2.toString());
// Prints: this is a tést
console.log(buf1.toString('ascii'));
// Prints: this is a tC)st
```

Class Method: Buffer.alloc(size[, fill[, encoding]])

▸ History

- `size` <integer> The desired length of the new `Buffer`.
- `fill` <string> | <Buffer> | <integer> A value to pre-fill the new `Buffer` with. **Default:** `0`
- `encoding` <string> If `fill` is a string, this is its encoding. **Default:** `'utf8'`

Allocates a new `Buffer` of `size` bytes. If `fill` is `undefined`, the `Buffer` will be *zero-filled*.

```
const buf = Buffer.alloc(5);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00>
```

Allocates a new `Buffer` of `size` bytes. If the `size` is larger than `buffer.constants.MAX_LENGTH` or smaller than 0, a `RangeError` will be thrown. A zero-length `Buffer` will be created if `size` is 0.

If `fill` is specified, the allocated `Buffer` will be initialized by calling `buf.fill(fill)`.

```
const buf = Buffer.alloc(5, 'a');

console.log(buf);
// Prints: <Buffer 61 61 61 61 61>
```

If both `fill` and `encoding` are specified, the allocated `Buffer` will be initialized by calling `buf.fill(fill, encoding)`.

```
const buf = Buffer.alloc(11, 'aGVsbG8gd29ybGQ=', 'base64');

console.log(buf);
// Prints: <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>
```

Calling `Buffer.alloc()` can be significantly slower than the alternative `Buffer.allocUnsafe()` but ensures that the newly created `Buffer` instance contents will *never contain sensitive data*.

A `TypeError` will be thrown if `size` is not a number.

Class Method: Buffer.allocUnsafe(size)

▸ History

- `size` <integer> The desired length of the new `Buffer`.

Allocates a new `Buffer` of `size` bytes. If the `size` is larger than `buffer.constants.MAX_LENGTH` or smaller than 0, a `RangeError` will be thrown. A zero-length `Buffer` will be created if `size` is 0.

The underlying memory for `Buffer` instances created in this way is *not initialized*. The contents of the newly created `Buffer` are unknown and *may contain sensitive data*. Use `Buffer.alloc()` instead to initialize `Buffer` instances with zeroes.

```
const buf = Buffer.allocUnsafe(10);

console.log(buf);
// Prints: (contents may vary): <Buffer a0 8b 28 3f 01 00 00 00 50 32>

buf.fill(0);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00 00 00 00 00 00>
```

A `TypeError` will be thrown if `size` is not a number.

Note that the `Buffer` module pre-allocates an internal `Buffer` instance of size `Buffer.poolSize` that is used as a pool for the fast allocation of new `Buffer` instances created using `Buffer.allocUnsafe()` and the deprecated `new Buffer(size)` constructor only when `size` is less than or equal to `Buffer.poolSize >> 1` (floor of `Buffer.poolSize` divided by two).

Use of this pre-allocated internal memory pool is a key difference between calling `Buffer.alloc(size, fill)` vs. `Buffer.allocUnsafe(size).fill(fill)`. Specifically, `Buffer.alloc(size, fill)` will *never* use the internal `Buffer` pool, while `Buffer.allocUnsafe(size).fill(fill)` *will* use the internal `Buffer` pool if `size` is less than or equal to half `Buffer.poolSize`. The difference is subtle but can be important when an application requires the additional performance that `Buffer.allocUnsafe()` provides.

Class Method: Buffer.allocUnsafeSlow(size)

Added in: v5.12.0

- `size` <integer> The desired length of the new `Buffer`.

Allocates a new `Buffer` of `size` bytes. If the `size` is larger than `buffer.constants.MAX_LENGTH` or smaller than 0, a `RangeError` will be thrown. A zero-length `Buffer` will be created if `size` is 0.

The underlying memory for `Buffer` instances created in this way is *not initialized*. The contents of the newly created `Buffer` are unknown and *may contain sensitive data*. Use `buf.fill(0)` to initialize such `Buffer` instances with zeroes.

When using `Buffer.allocUnsafe()` to allocate new `Buffer` instances, allocations under 4KB are sliced from a single pre-allocated `Buffer`. This allows applications to avoid the garbage collection overhead of creating many individually allocated `Buffer` instances. This approach improves both performance and memory usage by eliminating the need to track and clean up as many persistent objects.

However, in the case where a developer may need to retain a small chunk of memory from a pool for an indeterminate amount of time, it may

be appropriate to create an un-pooled `Buffer` instance using `Buffer.allocUnsafeSlow()` and then copying out the relevant bits.

```
// Need to keep around a few small chunks of memory
const store = [];

socket.on('readable', () => {
  const data = socket.read();

  // Allocate for retained data
  const sb = Buffer.allocUnsafeSlow(10);

  // Copy the data into the new allocation
  data.copy(sb, 0, 0, 10);

  store.push(sb);
});
```

`Buffer.allocUnsafeSlow()` should be used only as a last resort after a developer has observed undue memory retention in their applications.

A `TypeError` will be thrown if `size` is not a number.

Class Method: Buffer.byteLength(string[, encoding])

▸ History

- `string` <string> | <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <SharedArrayBuffer> A value to calculate the length of.
- `encoding` <string> If `string` is a string, this is its encoding. **Default:** `'utf8'`
- Returns: <integer> The number of bytes contained within `string`.

Returns the actual byte length of a string. This is not the same as `String.prototype.length` since that returns the number of *characters* in a string.

For `'base64'` and `'hex'`, this function assumes valid input. For strings that contain non-Base64/Hex-encoded data (e.g. whitespace), the return value might be greater than the length of a `Buffer` created from the string.

```
const str = '\u00bd + \u00bc = \u00be';

console.log(`${str}: ${str.length} characters, ` +
            `${Buffer.byteLength(str, 'utf8')} bytes`);
// Prints: ½ + ¼ = ¾: 9 characters, 12 bytes
```

When `string` is a `Buffer` / DataView / TypedArray / ArrayBuffer / SharedArrayBuffer, the actual byte length is returned.

Class Method: Buffer.compare(buf1, buf2)

▸ History

- `buf1` <Buffer> | <Uint8Array>
- `buf2` <Buffer> | <Uint8Array>
- Returns: <integer>

Compares `buf1` to `buf2` typically for the purpose of sorting arrays of `Buffer` instances. This is equivalent to calling `buf1.compare(buf2)`.

```
const buf1 = Buffer.from('1234');
const buf2 = Buffer.from('0123');
const arr = [buf1, buf2];

console.log(arr.sort(Buffer.compare));
// Prints: [ <Buffer 30 31 32 33>, <Buffer 31 32 33 34> ]
// (This result is equal to: [buf2, buf1])
```

Class Method: Buffer.concat(list[, totalLength])

▸ History

- `list` <Array> List of `Buffer` or `Uint8Array` instances to concat.
- `totalLength` <integer> Total length of the `Buffer` instances in `list` when concatenated.
- Returns: <Buffer>

Returns a new `Buffer` which is the result of concatenating all the `Buffer` instances in the `list` together.

If the list has no items, or if the `totalLength` is 0, then a new zero-length `Buffer` is returned.

If `totalLength` is not provided, it is calculated from the `Buffer` instances in `list`. This however causes an additional loop to be executed in order to calculate the `totalLength`, so it is faster to provide the length explicitly if it is already known.

If `totalLength` is provided, it is coerced to an unsigned integer. If the combined length of the `Buffer`s in `list` exceeds `totalLength`, the result is truncated to `totalLength`.

```
// Create a single `Buffer` from a list of three `Buffer` instances.

const buf1 = Buffer.alloc(10);
const buf2 = Buffer.alloc(14);
const buf3 = Buffer.alloc(18);
const totalLength = buf1.length + buf2.length + buf3.length;

console.log(totalLength);
// Prints: 42

const bufA = Buffer.concat([buf1, buf2, buf3], totalLength);

console.log(bufA);
// Prints: <Buffer 00 00 00 00 ...>
console.log(bufA.length);
// Prints: 42
```

Class Method: Buffer.from(array)

Added in: v5.10.0

- `array` <Array>

Allocates a new `Buffer` using an `array` of octets.

```
// Creates a new Buffer containing UTF-8 bytes of the string 'buffer'
const buf = Buffer.from([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);
```

A `TypeError` will be thrown if `array` is not an `Array`.

Class Method: Buffer.from(arrayBuffer[, byteOffset[, length]])

Added in: v5.10.0

- arrayBuffer <ArrayBuffer> | <SharedArrayBuffer> An ArrayBuffer , SharedArrayBuffer , or the .buffer property of a TypedArray .
- byteOffset <integer> Index of first byte to expose. **Default:** 0
- length <integer> Number of bytes to expose. **Default:** arrayBuffer.length - byteOffset

This creates a view of the ArrayBuffer without copying the underlying memory. For example, when passed a reference to the .buffer property of a TypedArray instance, the newly created Buffer will share the same allocated memory as the TypedArray .

```
const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Shares memory with `arr`
const buf = Buffer.from(arr.buffer);

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// Changing the original Uint16Array changes the Buffer also
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

The optional byteOffset and length arguments specify a memory range within the arrayBuffer that will be shared by the Buffer .

```
const ab = new ArrayBuffer(10);
const buf = Buffer.from(ab, 0, 2);

console.log(buf.length);
// Prints: 2
```

A TypeError will be thrown if arrayBuffer is not an ArrayBuffer or a SharedArrayBuffer .

Class Method: Buffer.from(buffer)

Added in: v5.10.0

- buffer <Buffer> An existing Buffer to copy data from.

Copies the passed buffer data onto a new Buffer instance.

```
const buf1 = Buffer.from('buffer');
const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;

console.log(buf1.toString());
// Prints: auffer
console.log(buf2.toString());
// Prints: buffer
```

A TypeError will be thrown if buffer is not a Buffer .

Class Method: Buffer.from(string[, encoding])

Added in: v5.10.0

- `string` <string> A string to encode.
- `encoding` <string> The encoding of `string`. **Default:** `'utf8'`

Creates a new `Buffer` containing `string`. The `encoding` parameter identifies the character encoding of `string`.

```
const buf1 = Buffer.from('this is a tést');
const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');

console.log(buf1.toString());
// Prints: this is a tést
console.log(buf2.toString());
// Prints: this is a tést
console.log(buf1.toString('ascii'));
// Prints: this is a tC)st
```

A `TypeError` will be thrown if `string` is not a string.

Class Method: Buffer.from(object[, offsetOrEncoding[, length]])

Added in: v8.2.0

- `object` <Object> An object supporting `Symbol.toPrimitive` or `valueOf()`
- `offsetOrEncoding` <number> | <string> A byte-offset or encoding, depending on the value returned either by `object.valueOf()` or `object[Symbol.toPrimitive]()`.
- `length` <number> A length, depending on the value returned either by `object.valueOf()` or `object[Symbol.toPrimitive]()`.

For objects whose `valueOf()` function returns a value not strictly equal to `object`, returns `Buffer.from(object.valueOf(), offsetOrEncoding, length)`.

```
const buf = Buffer.from(new String('this is a test'));
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>
```

For objects that support `Symbol.toPrimitive`, returns `Buffer.from(object[Symbol.toPrimitive](), offsetOrEncoding, length)`.

```
class Foo {
  [Symbol.toPrimitive]() {
    return 'this is a test';
  }
}

const buf = Buffer.from(new Foo(), 'utf8');
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>
```

Class Method: Buffer.isBuffer(obj)

Added in: v0.1.101

- `obj` <Object>
- Returns: <boolean>

Returns `true` if `obj` is a `Buffer`, `false` otherwise.

Class Method: Buffer.isEncoding(encoding)

Added in: v0.9.1

- `encoding` <string> A character encoding name to check.
- Returns: <boolean>

Returns `true` if `encoding` contains a supported character encoding, or `false` otherwise.

Class Property: Buffer.poolSize

Added in: v0.11.3

- <integer> **Default:** `8192`

This is the number of bytes used to determine the size of pre-allocated, internal `Buffer` instances used for pooling. This value may be modified.

buf[index]

The index operator `[index]` can be used to get and set the octet at position `index` in `buf`. The values refer to individual bytes, so the legal value range is between `0x00` and `0xFF` (hex) or `0` and `255` (decimal).

This operator is inherited from `Uint8Array`, so its behavior on out-of-bounds access is the same as `UInt8Array` - that is, getting returns `undefined` and setting does nothing.

```
// Copy an ASCII string into a `Buffer` one byte at a time.

const str = 'Node.js';
const buf = Buffer.allocUnsafe(str.length);

for (let i = 0; i < str.length; i++) {
  buf[i] = str.charCodeAt(i);
}

console.log(buf.toString('ascii'));
// Prints: Node.js
```

buf.buffer

The `buffer` property references the underlying `ArrayBuffer` object based on which this Buffer object is created.

```
const arrayBuffer = new ArrayBuffer(16);
const buffer = Buffer.from(arrayBuffer);

console.log(buffer.buffer === arrayBuffer);
// Prints: true
```

buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])

▸ History

- `target` <Buffer> | <Uint8Array> A `Buffer` or Uint8Array to compare to.
- `targetStart` <integer> The offset within `target` at which to begin comparison. **Default:** `0`
- `targetEnd` <integer> The offset with `target` at which to end comparison (not inclusive). **Default:** `target.length`
- `sourceStart` <integer> The offset within `buf` at which to begin comparison. **Default:** `0`
- `sourceEnd` <integer> The offset within `buf` at which to end comparison (not inclusive). **Default:** buf.length
- Returns: <integer>

Compares `buf` with `target` and returns a number indicating whether `buf` comes before, after, or is the same as `target` in sort order. Comparison is based on the actual sequence of bytes in each `Buffer`.

- `0` is returned if `target` is the same as `buf`
- `1` is returned if `target` should come *before* `buf` when sorted.
- `-1` is returned if `target` should come *after* `buf` when sorted.

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('BCD');
const buf3 = Buffer.from('ABCD');

console.log(buf1.compare(buf1));
// Prints: 0
console.log(buf1.compare(buf2));
// Prints: -1
console.log(buf1.compare(buf3));
// Prints: -1
console.log(buf2.compare(buf1));
// Prints: 1
console.log(buf2.compare(buf3));
// Prints: 1
console.log([buf1, buf2, buf3].sort(Buffer.compare));
// Prints: [ <Buffer 41 42 43>, <Buffer 41 42 43 44>, <Buffer 42 43 44> ]
// (This result is equal to: [buf1, buf3, buf2])
```

The optional `targetStart`, `targetEnd`, `sourceStart`, and `sourceEnd` arguments can be used to limit the comparison to specific ranges within `target` and `buf` respectively.

```
const buf1 = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8, 9]);
const buf2 = Buffer.from([5, 6, 7, 8, 9, 1, 2, 3, 4]);

console.log(buf1.compare(buf2, 5, 9, 0, 4));
// Prints: 0
console.log(buf1.compare(buf2, 0, 6, 4));
// Prints: -1
console.log(buf1.compare(buf2, 5, 6, 5));
// Prints: 1
```

A `RangeError` will be thrown if: `targetStart < 0`, `sourceStart < 0`, `targetEnd > target.byteLength` or `sourceEnd > source.byteLength`.

buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])

Added in: v0.1.90

- `target` <Buffer> | <Uint8Array> A `Buffer` or Uint8Array to copy into.
- `targetStart` <integer> The offset within `target` at which to begin copying to. **Default:** `0`
- `sourceStart` <integer> The offset within `buf` at which to begin copying from. **Default:** `0`
- `sourceEnd` <integer> The offset within `buf` at which to stop copying (not inclusive). **Default:** buf.length
- Returns: <integer> The number of bytes copied.

Copies data from a region of `buf` to a region in `target` even if the `target` memory region overlaps with `buf`.

```
// Create two `Buffer` instances.
const buf1 = Buffer.allocUnsafe(26);
const buf2 = Buffer.allocUnsafe(26).fill('!');

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'
  buf1[i] = i + 97;
}

// Copy `buf1` bytes 16 through 19 into `buf2` starting at byte 8 of `buf2`
buf1.copy(buf2, 8, 16, 20);

console.log(buf2.toString('ascii', 0, 25));
// Prints: !!!!!!!!qrst!!!!!!!!!!!!!
```

```
// Create a `Buffer` and copy data from one region to an overlapping region
// within the same `Buffer`.

const buf = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'
  buf[i] = i + 97;
}

buf.copy(buf, 0, 4, 10);

console.log(buf.toString());
// Prints: efghijghijklmnopqrstuvwxyz
```

buf.entries()

Added in: v1.1.0

- Returns: <Iterator>

Creates and returns an iterator of `[index, byte]` pairs from the contents of `buf`.

```
// Log the entire contents of a `Buffer`.

const buf = Buffer.from('buffer');

for (const pair of buf.entries()) {
  console.log(pair);
}
// Prints:
//   [0, 98]
//   [1, 117]
//   [2, 102]
//   [3, 102]
//   [4, 101]
//   [5, 114]
```

buf.equals(otherBuffer)

▸ History

- `otherBuffer` <Buffer> A `Buffer` or `Uint8Array` to compare to.
- Returns: <boolean>

Returns `true` if both `buf` and `otherBuffer` have exactly the same bytes, `false` otherwise.

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
const buf3 = Buffer.from('ABCD');

console.log(buf1.equals(buf2));
// Prints: true
console.log(buf1.equals(buf3));
// Prints: false
```

buf.fill(value[, offset[, end]][, encoding])

▸ History

- `value` <string> | <Buffer> | <integer> The value to fill `buf` with.
- `offset` <integer> Number of bytes to skip before starting to fill `buf`. **Default:** `0`
- `end` <integer> Where to stop filling `buf` (not inclusive). **Default:** `buf.length`
- `encoding` <string> If `value` is a string, this is its encoding. **Default:** `'utf8'`
- Returns: <Buffer> A reference to `buf`.

Fills `buf` with the specified `value`. If the `offset` and `end` are not given, the entire `buf` will be filled. This is meant to be a small simplification to allow the creation and filling of a `Buffer` to be done on a single line.

```
// Fill a `Buffer` with the ASCII character 'h'.

const b = Buffer.allocUnsafe(50).fill('h');

console.log(b.toString());
// Prints: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
```

`value` is coerced to a `uint32` value if it is not a String or Integer.

If the final write of a `fill()` operation falls on a multi-byte character, then only the first bytes of that character that fit into `buf` are written.

```
// Fill a `Buffer` with a two-byte character.

console.log(Buffer.allocUnsafe(3).fill('\u0222'));
// Prints: <Buffer c8 a2 c8>
```

If `value` contains invalid characters, it is truncated.

If no valid fill data remains, then the buffer is either zero-filled or no filling is performed, depending on the input type. That behavior is dictated by compatibility reasons and was changed to throwing an exception in Node.js v10, so it's not recommended to rely on that.

```js
const buf = Buffer.allocUnsafe(5);

console.log(buf.fill('a'));
// Prints: <Buffer 61 61 61 61 61>
console.log(buf.fill('aazz', 'hex'));
// Prints: <Buffer aa aa aa aa aa>
console.log(buf.fill('zz', 'hex'));
// Throws an exception.
```

buf.includes(value[, byteOffset][, encoding])

Added in: v5.3.0

- `value` <string> | <Buffer> | <integer> What to search for.
- `byteOffset` <integer> Where to begin searching in `buf`. **Default:** `0`
- `encoding` <string> If `value` is a string, this is its encoding. **Default:** `'utf8'`
- Returns: <boolean> `true` if `value` was found in `buf`, `false` otherwise.

Equivalent to `buf.indexOf() !== -1`.

```js
const buf = Buffer.from('this is a buffer');

console.log(buf.includes('this'));
// Prints: true
console.log(buf.includes('is'));
// Prints: true
console.log(buf.includes(Buffer.from('a buffer')));
// Prints: true
console.log(buf.includes(97));
// Prints: true (97 is the decimal ASCII value for 'a')
console.log(buf.includes(Buffer.from('a buffer example')));
// Prints: false
console.log(buf.includes(Buffer.from('a buffer example').slice(0, 8)));
// Prints: true
console.log(buf.includes('this', 4));
// Prints: false
```

buf.indexOf(value[, byteOffset][, encoding])

▸ History

- `value` <string> | <Buffer> | <Uint8Array> | <integer> What to search for.
- `byteOffset` <integer> Where to begin searching in `buf`. **Default:** `0`
- `encoding` <string> If `value` is a string, this is its encoding. **Default:** `'utf8'`
- Returns: <integer> The index of the first occurrence of `value` in `buf` or `-1` if `buf` does not contain `value`.

If `value` is:

- a string, `value` is interpreted according to the character encoding in `encoding`.
- a `Buffer` or `Uint8Array`, `value` will be used in its entirety. To compare a partial `Buffer`, use `buf.slice()`.
- a number, `value` will be interpreted as an unsigned 8-bit integer value between `0` and `255`.

```js
const buf = Buffer.from('this is a buffer');

console.log(buf.indexOf('this'));
// Prints: 0
console.log(buf.indexOf('is'));
// Prints: 2
console.log(buf.indexOf(Buffer.from('a buffer')));
// Prints: 8
console.log(buf.indexOf(97));
// Prints: 8 (97 is the decimal ASCII value for 'a')
console.log(buf.indexOf(Buffer.from('a buffer example')));
// Prints: -1
console.log(buf.indexOf(Buffer.from('a buffer example').slice(0, 8)));
// Prints: 8

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.indexOf('\u03a3', 0, 'utf16le'));
// Prints: 4
console.log(utf16Buffer.indexOf('\u03a3', -4, 'utf16le'));
// Prints: 6
```

If `value` is not a string, number, or `Buffer`, this method will throw a `TypeError`. If `value` is a number, it will be coerced to a valid byte value, an integer between 0 and 255.

If `byteOffset` is not a number, it will be coerced to a number. Any arguments that coerce to `NaN` or 0, like `{}`, `[]`, `null` or `undefined`, will search the whole buffer. This behavior matches String#indexOf() .

```js
const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte
// Prints: 2, equivalent to searching for 99 or 'c'
console.log(b.indexOf(99.9));
console.log(b.indexOf(256 + 99));

// Passing a byteOffset that coerces to NaN or 0
// Prints: 1, searching the whole buffer
console.log(b.indexOf('b', undefined));
console.log(b.indexOf('b', {}));
console.log(b.indexOf('b', null));
console.log(b.indexOf('b', []));
```

If `value` is an empty string or empty `Buffer` and `byteOffset` is less than `buf.length`, `byteOffset` will be returned. If `value` is empty and `byteOffset` is at least `buf.length`, `buf.length` will be returned.

buf.keys()

Added in: v1.1.0

- Returns: <Iterator>

Creates and returns an iterator of `buf` keys (indices).

```js
const buf = Buffer.from('buffer');

for (const key of buf.keys()) {
  console.log(key);
}
// Prints:
//   0
//   1
//   2
//   3
//   4
//   5
```

buf.lastIndexOf(value[, byteOffset][, encoding])

▸ History

- `value` <string> | <Buffer> | <Uint8Array> | <integer> What to search for.
- `byteOffset` <integer> Where to begin searching in `buf`. **Default:** `buf.length - 1`
- `encoding` <string> If `value` is a string, this is its encoding. **Default:** `'utf8'`
- Returns: <integer> The index of the last occurrence of `value` in `buf` or `-1` if `buf` does not contain `value`.

Identical to `buf.indexOf()`, except `buf` is searched from back to front instead of front to back.

```js
const buf = Buffer.from('this buffer is a buffer');

console.log(buf.lastIndexOf('this'));
// Prints: 0
console.log(buf.lastIndexOf('buffer'));
// Prints: 17
console.log(buf.lastIndexOf(Buffer.from('buffer')));
// Prints: 17
console.log(buf.lastIndexOf(97));
// Prints: 15 (97 is the decimal ASCII value for 'a')
console.log(buf.lastIndexOf(Buffer.from('yolo')));
// Prints: -1
console.log(buf.lastIndexOf('buffer', 5));
// Prints: 5
console.log(buf.lastIndexOf('buffer', 4));
// Prints: -1

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.lastIndexOf('\u03a3', undefined, 'utf16le'));
// Prints: 6
console.log(utf16Buffer.lastIndexOf('\u03a3', -5, 'utf16le'));
// Prints: 4
```

If `value` is not a string, number, or `Buffer`, this method will throw a `TypeError`. If `value` is a number, it will be coerced to a valid byte value, an integer between 0 and 255.

If `byteOffset` is not a number, it will be coerced to a number. Any arguments that coerce to `NaN`, like `{}` or `undefined`, will search the whole buffer. This behavior matches `String#lastIndexOf()`.

```js
const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte
// Prints: 2, equivalent to searching for 99 or 'c'
console.log(b.lastIndexOf(99.9));
console.log(b.lastIndexOf(256 + 99));

// Passing a byteOffset that coerces to NaN
// Prints: 1, searching the whole buffer
console.log(b.lastIndexOf('b', undefined));
console.log(b.lastIndexOf('b', {}));

// Passing a byteOffset that coerces to 0
// Prints: -1, equivalent to passing 0
console.log(b.lastIndexOf('b', null));
console.log(b.lastIndexOf('b', []));
```

If `value` is an empty string or empty `Buffer`, `byteOffset` will be returned.

buf.length

Added in: v0.1.90

- <integer>

Returns the amount of memory allocated for `buf` in bytes. Note that this does not necessarily reflect the amount of "usable" data within `buf`.

```js
// Create a `Buffer` and write a shorter ASCII string to it.

const buf = Buffer.alloc(1234);

console.log(buf.length);
// Prints: 1234

buf.write('some string', 0, 'ascii');

console.log(buf.length);
// Prints: 1234
```

While the `length` property is not immutable, changing the value of `length` can result in undefined and inconsistent behavior. Applications that wish to modify the length of a `Buffer` should therefore treat `length` as read-only and use buf.slice() to create a new `Buffer`.

```js
let buf = Buffer.allocUnsafe(10);

buf.write('abcdefghj', 0, 'ascii');

console.log(buf.length);
// Prints: 10

buf = buf.slice(0, 5);

console.log(buf.length);
// Prints: 5
```

buf.parent

Deprecated since: v8.0.0

Stability: 0 - Deprecated: Use buf.buffer instead.

The `buf.parent` property is a deprecated alias for `buf.buffer`.

buf.readDoubleBE(offset[, noAssert])

buf.readDoubleLE(offset[, noAssert])

Added in: v0.11.15

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <number>

Reads a 64-bit double from `buf` at the specified `offset` with specified endian format (`readDoubleBE()` returns big endian, `readDoubleLE()` returns little endian).

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleBE());
// Prints: 8.20788039913184e-304
console.log(buf.readDoubleLE());
// Prints: 5.447603722011605e-270
console.log(buf.readDoubleLE(1));
// Throws an exception: RangeError: Index out of range
console.log(buf.readDoubleLE(1, true));
// Warning: reads passed end of buffer!
// This will result in a segmentation fault! Don't do this!
```

buf.readFloatBE(offset[, noAssert])

buf.readFloatLE(offset[, noAssert])

Added in: v0.11.15

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 4`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <number>

Reads a 32-bit float from `buf` at the specified `offset` with specified endian format (`readFloatBE()` returns big endian, `readFloatLE()` returns little endian).

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatBE());
// Prints: 2.387939260590663e-38
console.log(buf.readFloatLE());
// Prints: 1.539989614439558e-36
console.log(buf.readFloatLE(1));
// Throws an exception: RangeError: Index out of range
console.log(buf.readFloatLE(1, true));
// Warning: reads passed end of buffer!
// This will result in a segmentation fault! Don't do this!
```

buf.readInt8(offset[, noAssert])

Added in: v0.5.0

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 1`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <integer>

Reads a signed 8-bit integer from `buf` at the specified `offset`.

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
const buf = Buffer.from([-1, 5]);

console.log(buf.readInt8(0));
// Prints: -1
console.log(buf.readInt8(1));
// Prints: 5
console.log(buf.readInt8(2));
// Throws an exception: RangeError: Index out of range
```

buf.readInt16BE(offset[, noAssert])

buf.readInt16LE(offset[, noAssert])

Added in: v0.5.5

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 2`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <integer>

Reads a signed 16-bit integer from `buf` at the specified `offset` with the specified endian format (`readInt16BE()` returns big endian, `readInt16LE()` returns little endian).

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```js
const buf = Buffer.from([0, 5]);

console.log(buf.readInt16BE());
// Prints: 5
console.log(buf.readInt16LE());
// Prints: 1280
console.log(buf.readInt16LE(1));
// Throws an exception: RangeError: Index out of range
```

buf.readInt32BE(offset[, noAssert])

buf.readInt32LE(offset[, noAssert])

Added in: v0.5.5

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 4`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <integer>

Reads a signed 32-bit integer from `buf` at the specified `offset` with the specified endian format (`readInt32BE()` returns big endian, `readInt32LE()` returns little endian).

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```js
const buf = Buffer.from([0, 0, 0, 5]);

console.log(buf.readInt32BE());
// Prints: 5
console.log(buf.readInt32LE());
// Prints: 83886080
console.log(buf.readInt32LE(1));
// Throws an exception: RangeError: Index out of range
```

buf.readIntBE(offset, byteLength[, noAssert])

buf.readIntLE(offset, byteLength[, noAssert])

Added in: v0.11.15

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to read. Must satisfy: `0 < byteLength <= 6`.
- `noAssert` <boolean> Skip `offset` and `byteLength` validation? **Default:** `false`.
- Returns: <integer>

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as a two's complement signed value. Supports up to 48 bits of accuracy.

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntLE(0, 6).toString(16));
// Prints: -546f87a9cbee
console.log(buf.readIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readIntBE(1, 6).toString(16));
// Throws: RangeError [ERR_INDEX_OUT_OF_RANGE]: Index out of range
```

buf.readUInt8(offset[, noAssert])

Added in: v0.5.0

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 1`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <integer>

Reads an unsigned 8-bit integer from `buf` at the specified `offset`.

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.from([1, -2]);

console.log(buf.readUInt8(0));
// Prints: 1
console.log(buf.readUInt8(1));
// Prints: 254
console.log(buf.readUInt8(2));
// Throws an exception: RangeError: Index out of range
```

buf.readUInt16BE(offset[, noAssert])

buf.readUInt16LE(offset[, noAssert])

Added in: v0.5.5

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 2`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <integer>

Reads an unsigned 16-bit integer from `buf` at the specified `offset` with specified endian format (`readUInt16BE()` returns big endian, `readUInt16LE()` returns little endian).

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.from([0x12, 0x34, 0x56]);

console.log(buf.readUInt16BE(0).toString(16));
// Prints: 1234
console.log(buf.readUInt16LE(0).toString(16));
// Prints: 3412
console.log(buf.readUInt16BE(1).toString(16));
// Prints: 3456
console.log(buf.readUInt16LE(1).toString(16));
// Prints: 5634
console.log(buf.readUInt16LE(2).toString(16));
// Throws an exception: RangeError: Index out of range
```

buf.readUInt32BE(offset[, noAssert])

buf.readUInt32LE(offset[, noAssert])

Added in: v0.5.5

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 4`.
- `noAssert` <boolean> Skip `offset` validation? **Default:** `false`
- Returns: <integer>

Reads an unsigned 32-bit integer from `buf` at the specified `offset` with specified endian format (`readUInt32BE()` returns big endian, `readUInt32LE()` returns little endian).

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);

console.log(buf.readUInt32BE(0).toString(16));
// Prints: 12345678
console.log(buf.readUInt32LE(0).toString(16));
// Prints: 78563412
console.log(buf.readUInt32LE(1).toString(16));
// Throws an exception: RangeError: Index out of range
```

buf.readUIntBE(offset, byteLength[, noAssert])

buf.readUIntLE(offset, byteLength[, noAssert])

Added in: v0.11.15

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to read. Must satisfy: `0 < byteLength <= 6`.
- `noAssert` <boolean> Skip `offset` and `byteLength` validation? **Default:** `false`
- Returns: <integer>

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as an unsigned integer. Supports up to 48 bits of accuracy.

Setting `noAssert` to `true` allows `offset` to be beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readUIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readUIntLE(0, 6).toString(16));
// Prints: ab9078563412
console.log(buf.readUIntBE(1, 6).toString(16));
// Throws an exception: RangeError: Index out of range
```

buf.slice([start[, end]])

▸ History

- `start` <integer> Where the new `Buffer` will start. **Default:** `0`
- `end` <integer> Where the new `Buffer` will end (not inclusive). **Default:** buf.length
- Returns: <Buffer>

Returns a new `Buffer` that references the same memory as the original, but offset and cropped by the `start` and `end` indices.

Specifying `end` greater than buf.length will return the same result as that of `end` equal to buf.length.

Modifying the new `Buffer` slice will modify the memory in the original `Buffer` because the allocated memory of the two objects overlap.

```
// Create a `Buffer` with the ASCII alphabet, take a slice, and modify one byte
// from the original `Buffer`.

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'
  buf1[i] = i + 97;
}

const buf2 = buf1.slice(0, 3);

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: abc

buf1[0] = 33;

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: !bc
```

Specifying negative indexes causes the slice to be generated relative to the end of `buf` rather than the beginning.

```
const buf = Buffer.from('buffer');

console.log(buf.slice(-6, -1).toString());
// Prints: buffe
// (Equivalent to buf.slice(0, 5))

console.log(buf.slice(-6, -2).toString());
// Prints: buff
// (Equivalent to buf.slice(0, 4))

console.log(buf.slice(-5, -2).toString());
// Prints: uff
// (Equivalent to buf.slice(1, 4))
```

buf.swap16()

Added in: v5.10.0

- Returns: <Buffer> A reference to `buf`.

Interprets `buf` as an array of unsigned 16-bit integers and swaps the byte-order *in-place*. Throws a `RangeError` if `buf.length` is not a multiple of 2.

```
const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap16();

console.log(buf1);
// Prints: <Buffer 02 01 04 03 06 05 08 07>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap16();
// Throws an exception: RangeError: Buffer size must be a multiple of 16-bits
```

buf.swap32()

Added in: v5.10.0

- Returns: <Buffer> A reference to `buf`.

Interprets `buf` as an array of unsigned 32-bit integers and swaps the byte-order *in-place*. Throws a `RangeError` if `buf.length` is not a multiple of 4.

```
const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap32();

console.log(buf1);
// Prints: <Buffer 04 03 02 01 08 07 06 05>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap32();
// Throws an exception: RangeError: Buffer size must be a multiple of 32-bits
```

buf.swap64()

Added in: v6.3.0

- Returns: <Buffer> A reference to `buf`.

Interprets `buf` as an array of 64-bit numbers and swaps the byte-order *in-place*. Throws a `RangeError` if buf.length is not a multiple of 8.

```
const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap64();

console.log(buf1);
// Prints: <Buffer 08 07 06 05 04 03 02 01>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap64();
// Throws an exception: RangeError: Buffer size must be a multiple of 64-bits
```

Note that JavaScript cannot encode 64-bit integers. This method is intended for working with 64-bit floats.

buf.toJSON()

Added in: v0.9.2

- Returns: <Object>

Returns a JSON representation of `buf`. JSON.stringify() implicitly calls this function when stringifying a `Buffer` instance.

```js
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
const json = JSON.stringify(buf);

console.log(json);
// Prints: {"type":"Buffer","data":[1,2,3,4,5]}

const copy = JSON.parse(json, (key, value) => {
  return value && value.type === 'Buffer' ?
    Buffer.from(value.data) :
    value;
});

console.log(copy);
// Prints: <Buffer 01 02 03 04 05>
```

buf.toString([encoding[, start[, end]]])

Added in: v0.1.90

- `encoding` <string> The character encoding to decode to. **Default:** `'utf8'`
- `start` <integer> The byte offset to start decoding at. **Default:** `0`
- `end` <integer> The byte offset to stop decoding at (not inclusive). **Default:** buf.length
- Returns: <string>

Decodes `buf` to a string according to the specified character encoding in `encoding`. `start` and `end` may be passed to decode only a subset of `buf`.

The maximum length of a string instance (in UTF-16 code units) is available as buffer.constants.MAX_STRING_LENGTH .

```js
const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'
  buf1[i] = i + 97;
}

console.log(buf1.toString('ascii'));
// Prints: abcdefghijklmnopqrstuvwxyz
console.log(buf1.toString('ascii', 0, 5));
// Prints: abcde

const buf2 = Buffer.from('tést');

console.log(buf2.toString('hex'));
// Prints: 74c3a97374
console.log(buf2.toString('utf8', 0, 3));
// Prints: té
console.log(buf2.toString(undefined, 0, 3));
// Prints: té
```

buf.values()

Added in: v1.1.0

- Returns: <Iterator>

Creates and returns an iterator for `buf` values (bytes). This function is called automatically when a `Buffer` is used in a `for..of` statement.

```js
const buf = Buffer.from('buffer');

for (const value of buf.values()) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
//   102
//   101
//   114

for (const value of buf) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
//   102
//   101
//   114
```

buf.write(string[, offset[, length]][, encoding])

Added in: v0.1.90

- `string` <string> String to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write `string`. **Default:** `0`
- `length` <integer> Number of bytes to write. **Default:** `buf.length - offset`
- `encoding` <string> The character encoding of `string`. **Default:** `'utf8'`
- Returns: <integer> Number of bytes written.

Writes `string` to `buf` at `offset` according to the character encoding in `encoding`. The `length` parameter is the number of bytes to write. If `buf` did not contain enough space to fit the entire string, only a partial amount of `string` will be written. However, partially encoded characters will not be written.

```js
const buf = Buffer.allocUnsafe(256);

const len = buf.write('\u00bd + \u00bc = \u00be', 0);

console.log(`${len} bytes: ${buf.toString('utf8', 0, len)}`);
// Prints: 12 bytes: ½ + ¼ = ¾
```

buf.writeDoubleBE(value, offset[, noAssert])

buf.writeDoubleLE(value, offset[, noAssert])

Added in: v0.11.15

- `value` <number> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`

- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` with specified endian format (`writeDoubleBE()` writes big endian, `writeDoubleLE()` writes little endian). `value` *should* be a valid 64-bit double. Behavior is undefined when `value` is anything other than a 64-bit double.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.allocUnsafe(8);

buf.writeDoubleBE(0xdeadbeefcafebabe, 0);

console.log(buf);
// Prints: <Buffer 43 eb d5 b7 dd f9 5f d7>

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

console.log(buf);
// Prints: <Buffer d7 5f f9 dd b7 d5 eb 43>
```

buf.writeFloatBE(value, offset[, noAssert])

buf.writeFloatLE(value, offset[, noAssert])

Added in: v0.11.15

- `value` <number> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 4`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` with specified endian format (`writeFloatBE()` writes big endian, `writeFloatLE()` writes little endian). `value` *should* be a valid 32-bit float. Behavior is undefined when `value` is anything other than a 32-bit float.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.allocUnsafe(4);

buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer 4f 4a fe bb>

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer bb fe 4a 4f>
```

buf.writeInt8(value, offset[, noAssert])

Added in: v0.5.0

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 1`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset`. `value` *should* be a valid signed 8-bit integer. Behavior is undefined when `value` is anything other

than a signed 8-bit integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

`value` is interpreted and written as a two's complement signed integer.

```
const buf = Buffer.allocUnsafe(2);

buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);

console.log(buf);
// Prints: <Buffer 02 fe>
```

buf.writeInt16BE(value, offset[, noAssert])

buf.writeInt16LE(value, offset[, noAssert])

Added in: v0.5.5

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 2`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` with specified endian format (`writeInt16BE()` writes big endian, `writeInt16LE()` writes little endian). `value` *should* be a valid signed 16-bit integer. Behavior is undefined when `value` is anything other than a signed 16-bit integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

`value` is interpreted and written as a two's complement signed integer.

```
const buf = Buffer.allocUnsafe(4);

buf.writeInt16BE(0x0102, 0);
buf.writeInt16LE(0x0304, 2);

console.log(buf);
// Prints: <Buffer 01 02 04 03>
```

buf.writeInt32BE(value, offset[, noAssert])

buf.writeInt32LE(value, offset[, noAssert])

Added in: v0.5.5

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 4`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` with specified endian format (`writeInt32BE()` writes big endian, `writeInt32LE()` writes little endian). `value` *should* be a valid signed 32-bit integer. Behavior is undefined when `value` is anything other than a signed 32-bit integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

`value` is interpreted and written as a two's complement signed integer.

```js
const buf = Buffer.allocUnsafe(8);

buf.writeInt32BE(0x01020304, 0);
buf.writeInt32LE(0x05060708, 4);

console.log(buf);
// Prints: <Buffer 01 02 03 04 08 07 06 05>
```

buf.writeIntBE(value, offset, byteLength[, noAssert])

buf.writeIntLE(value, offset, byteLength[, noAssert])

Added in: v0.11.15

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to write. Must satisfy: `0 < byteLength <= 6`.
- `noAssert` <boolean> Skip `value`, `offset`, and `byteLength` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset`. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than a signed integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

```js
const buf = Buffer.allocUnsafe(6);

buf.writeIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>

buf.writeIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>
```

buf.writeUInt8(value, offset[, noAssert])

Added in: v0.5.0

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 1`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset`. `value` *should* be a valid unsigned 8-bit integer. Behavior is undefined when `value` is anything other than an unsigned 8-bit integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.allocUnsafe(4);

buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);
// Prints: <Buffer 03 04 23 42>
```

buf.writeUInt16BE(value, offset[, noAssert])

buf.writeUInt16LE(value, offset[, noAssert])

Added in: v0.5.5

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 2`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` with specified endian format (`writeUInt16BE()` writes big endian, `writeUInt16LE()` writes little endian). `value` should be a valid unsigned 16-bit integer. Behavior is undefined when `value` is anything other than an unsigned 16-bit integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

```
const buf = Buffer.allocUnsafe(4);

buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer de ad be ef>

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer ad de ef be>
```

buf.writeUInt32BE(value, offset[, noAssert])

buf.writeUInt32LE(value, offset[, noAssert])

Added in: v0.5.5

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 4`.
- `noAssert` <boolean> Skip `value` and `offset` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` with specified endian format (`writeUInt32BE()` writes big endian, `writeUInt32LE()` writes little endian). `value` should be a valid unsigned 32-bit integer. Behavior is undefined when `value` is anything other than an unsigned 32-bit integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

```js
const buf = Buffer.allocUnsafe(4);

buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer fe ed fa ce>

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer ce fa ed fe>
```

buf.writeUIntBE(value, offset, byteLength[, noAssert])

buf.writeUIntLE(value, offset, byteLength[, noAssert])

Added in: v0.5.5

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to write. Must satisfy: `0 < byteLength <= 6`.
- `noAssert` <boolean> Skip `value`, `offset`, and `byteLength` validation? **Default:** `false`
- Returns: <integer> `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset`. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than an unsigned integer.

Setting `noAssert` to `true` allows the encoded form of `value` to extend beyond the end of `buf`, but the resulting behavior is undefined.

```js
const buf = Buffer.allocUnsafe(6);

buf.writeUIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>

buf.writeUIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>
```

buffer.INSPECT_MAX_BYTES

Added in: v0.5.4

- <integer> **Default:** `50`

Returns the maximum number of bytes that will be returned when `buf.inspect()` is called. This can be overridden by user modules. See `util.inspect()` for more details on `buf.inspect()` behavior.

Note that this is a property on the `buffer` module returned by `require('buffer')`, not on the `Buffer` global or a `Buffer` instance.

buffer.kMaxLength

Added in: v3.0.0

- <integer> The largest size allowed for a single `Buffer` instance.

An alias for buffer.constants.MAX_LENGTH

Note that this is a property on the `buffer` module returned by `require('buffer')`, not on the `Buffer` global or a `Buffer` instance.

buffer.transcode(source, fromEnc, toEnc)

▸ History

- `source` <Buffer> | <Uint8Array> A `Buffer` or `Uint8Array` instance.
- `fromEnc` <string> The current encoding.
- `toEnc` <string> To target encoding.

Re-encodes the given `Buffer` or `Uint8Array` instance from one character encoding to another. Returns a new `Buffer` instance.

Throws if the `fromEnc` or `toEnc` specify invalid character encodings or if conversion from `fromEnc` to `toEnc` is not permitted.

The transcoding process will use substitution characters if a given byte sequence cannot be adequately represented in the target encoding. For instance:

```
const buffer = require('buffer');

const newBuf = buffer.transcode(Buffer.from('€'), 'utf8', 'ascii');
console.log(newBuf.toString('ascii'));
// Prints: '?'
```

Because the Euro (`€`) sign is not representable in US-ASCII, it is replaced with `?` in the transcoded `Buffer`.

Note that this is a property on the `buffer` module returned by `require('buffer')`, not on the `Buffer` global or a `Buffer` instance.

Class: SlowBuffer

Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use Buffer.allocUnsafeSlow() instead.

Returns an un-pooled `Buffer`.

In order to avoid the garbage collection overhead of creating many individually allocated `Buffer` instances, by default allocations under 4KB are sliced from a single larger allocated object.

In the case where a developer may need to retain a small chunk of memory from a pool for an indeterminate amount of time, it may be appropriate to create an un-pooled `Buffer` instance using `SlowBuffer` then copy out the relevant bits.

```
// Need to keep around a few small chunks of memory
const store = [];

socket.on('readable', () => {
  const data = socket.read();

  // Allocate for retained data
  const sb = SlowBuffer(10);

  // Copy the data into the new allocation
  data.copy(sb, 0, 0, 10);

  store.push(sb);
});
```

Use of `SlowBuffer` should be used only as a last resort *after* a developer has observed undue memory retention in their applications.

new SlowBuffer(size)

Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use Buffer.allocUnsafeSlow() instead.

- `size` <integer> The desired length of the new `SlowBuffer`.

Allocates a new `Buffer` of `size` bytes. If the `size` is larger than buffer.constants.MAX_LENGTH or smaller than 0, a RangeError will be thrown. A zero-length `Buffer` will be created if `size` is 0.

The underlying memory for `SlowBuffer` instances is *not initialized*. The contents of a newly created `SlowBuffer` are unknown and may contain sensitive data. Use buf.fill(0) to initialize a `SlowBuffer` with zeroes.

```
const { SlowBuffer } = require('buffer');

const buf = new SlowBuffer(5);

console.log(buf);
// Prints: (contents may vary): <Buffer 78 e0 82 02 01>

buf.fill(0);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00>
```

Buffer Constants

Added in: v8.2.0

Note that `buffer.constants` is a property on the `buffer` module returned by `require('buffer')`, not on the `Buffer` global or a `Buffer` instance.

buffer.constants.MAX_LENGTH

Added in: v8.2.0

- <integer> The largest size allowed for a single `Buffer` instance.

On 32-bit architectures, this value is `(2^30)-1` (~1GB). On 64-bit architectures, this value is `(2^31)-1` (~2GB).

This value is also available as buffer.kMaxLength.

buffer.constants.MAX_STRING_LENGTH

Added in: v8.2.0

- <integer> The largest length allowed for a single `string` instance.

Represents the largest `length` that a `string` primitive can have, counted in UTF-16 code units.

This value may depend on the JS engine that is being used.

C++ Addons

Node.js Addons are dynamically-linked shared objects, written in C++, that can be loaded into Node.js using the require() function, and used just as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries.

At the moment, the method for implementing Addons is rather complicated, involving knowledge of several components and APIs :

- V8: the C++ library Node.js currently uses to provide the JavaScript implementation. V8 provides the mechanisms for creating objects, calling functions, etc. V8's API is documented mostly in the `v8.h` header file (`deps/v8/include/v8.h` in the Node.js source tree), which is also available online .
- libuv : The C library that implements the Node.js event loop, its worker threads and all of the asynchronous behaviors of the platform. It also serves as a cross-platform abstraction library, giving easy, POSIX-like access across all major operating systems to many common system tasks, such as interacting with the filesystem, sockets, timers, and system events. libuv also provides a pthreads-like threading abstraction that may be used to power more sophisticated asynchronous Addons that need to move beyond the standard event loop. Addon authors are encouraged to think about how to avoid blocking the event loop with I/O or other time-intensive tasks by off-loading work via libuv to non-blocking system operations, worker threads or a custom use of libuv's threads.
- Internal Node.js libraries. Node.js itself exports a number of C++ APIs that Addons can use — the most important of which is the `node::ObjectWrap` class.
- Node.js includes a number of other statically linked libraries including OpenSSL. These other libraries are located in the `deps/` directory in the Node.js source tree. Only the libuv, OpenSSL, V8 and zlib symbols are purposefully re-exported by Node.js and may be used to various extents by Addons. See Linking to Node.js' own dependencies for additional information.

All of the following examples are available for download and may be used as the starting-point for an Addon.

Hello world

This "Hello world" example is a simple Addon, written in C++, that is the equivalent of the following JavaScript code:

```
module.exports.hello = () => 'world';
```

First, create the file `hello.cc`:

```
// hello.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"));
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, init)

}  // namespace demo
```

Note that all Node.js Addons must export an initialization function following the pattern:

```
void Initialize(Local<Object> exports);
NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

There is no semi-colon after `NODE_MODULE` as it's not a function (see `node.h`).

The `module_name` must match the filename of the final binary (excluding the .node suffix).

In the `hello.cc` example, then, the initialization function is `init` and the Addon module name is `addon`.

Building

Once the source code has been written, it must be compiled into the binary `addon.node` file. To do so, create a file called `binding.gyp` in the top-level of the project describing the build configuration of the module using a JSON-like format. This file is used by node-gyp — a tool written specifically to compile Node.js Addons.

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

A version of the `node-gyp` utility is bundled and distributed with Node.js as part of `npm`. This version is not made directly available for developers to use and is intended only to support the ability to use the `npm install` command to compile and install Addons. Developers who wish to use `node-gyp` directly can install it using the command `npm install -g node-gyp`. See the `node-gyp` installation instructions for more information, including platform-specific requirements.

Once the `binding.gyp` file has been created, use `node-gyp configure` to generate the appropriate project build files for the current platform. This will generate either a `Makefile` (on Unix platforms) or a `vcxproj` file (on Windows) in the `build/` directory.

Next, invoke the `node-gyp build` command to generate the compiled `addon.node` file. This will be put into the `build/Release/` directory.

When using `npm install` to install a Node.js Addon, npm uses its own bundled version of `node-gyp` to perform this same set of actions, generating a compiled version of the Addon for the user's platform on demand.

Once built, the binary Addon can be used from within Node.js by pointing `require()` to the built `addon.node` module:

```
// hello.js
const addon = require('./build/Release/addon');

console.log(addon.hello());
// Prints: 'world'
```

Please see the examples below for further information or https://github.com/arturadib/node-qt for an example in production.

Because the exact path to the compiled Addon binary can vary depending on how it is compiled (i.e. sometimes it may be in `./build/Debug/`), Addons can use the bindings package to load the compiled module.

Note that while the `bindings` package implementation is more sophisticated in how it locates Addon modules, it is essentially using a try-catch pattern similar to:

```
try {
  return require('./build/Release/addon.node');
} catch (err) {
  return require('./build/Debug/addon.node');
}
```

Linking to Node.js' own dependencies

Node.js uses a number of statically linked libraries such as V8, libuv and OpenSSL. All Addons are required to link to V8 and may link to any of the other dependencies as well. Typically, this is as simple as including the appropriate `#include <...>` statements (e.g. `#include <v8.h>`) and `node-gyp` will locate the appropriate headers automatically. However, there are a few caveats to be aware of:

- When `node-gyp` runs, it will detect the specific release version of Node.js and download either the full source tarball or just the headers. If the full source is downloaded, Addons will have complete access to the full set of Node.js dependencies. However, if only the Node.js headers are downloaded, then only the symbols exported by Node.js will be available.

- `node-gyp` can be run using the `--nodedir` flag pointing at a local Node.js source image. Using this option, the Addon will have access to the full set of dependencies.

Loading Addons using require()

The filename extension of the compiled Addon binary is `.node` (as opposed to `.dll` or `.so`). The `require()` function is written to look for files with the `.node` file extension and initialize those as dynamically-linked libraries.

When calling `require()`, the `.node` extension can usually be omitted and Node.js will still find and initialize the Addon. One caveat, however, is that Node.js will first attempt to locate and load modules or JavaScript files that happen to share the same base name. For instance, if there is a file `addon.js` in the same directory as the binary `addon.node`, then `require('addon')` will give precedence to the `addon.js` file and load it instead.

Native Abstractions for Node.js

Each of the examples illustrated in this document make direct use of the Node.js and V8 APIs for implementing Addons. It is important to understand that the V8 API can, and has, changed dramatically from one V8 release to the next (and one major Node.js release to the next). With each change, Addons may need to be updated and recompiled in order to continue functioning. The Node.js release schedule is designed to minimize the frequency and impact of such changes but there is little that Node.js can do currently to ensure stability of the V8 APIs.

The Native Abstractions for Node.js (or `nan`) provide a set of tools that Addon developers are recommended to use to keep compatibility between past and future releases of V8 and Node.js. See the `nan` examples for an illustration of how it can be used.

N-API

Stability: 1 - Experimental

N-API is an API for building native Addons. It is independent from the underlying JavaScript runtime (e.g. V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across version of Node.js. It is intended to insulate Addons from changes in the underlying JavaScript engine and allow modules compiled for one version to run on later versions of Node.js without recompilation. Addons are built/packaged with the same approach/tools outlined in this document (node-gyp, etc.). The only difference is the set of APIs that are used by the native code. Instead of using the V8 or Native Abstractions for Node.js APIs, the functions available in the N-API are used.

To use N-API in the above "Hello world" example, replace the content of `hello.cc` with the following. All other instructions remain the same.

```
// hello.cc using N-API
#include <node_api.h>

namespace demo {

napi_value Method(napi_env env, napi_callback_info args) {
  napi_value greeting;
  napi_status status;

  status = napi_create_string_utf8(env, "hello", NAPI_AUTO_LENGTH, &greeting);
  if (status != napi_ok) return nullptr;
  return greeting;
}

napi_value init(napi_env env, napi_value exports) {
  napi_status status;
  napi_value fn;

  status = napi_create_function(env, nullptr, 0, Method, nullptr, &fn);
  if (status != napi_ok) return nullptr;

  status = napi_set_named_property(env, exports, "hello", fn);
  if (status != napi_ok) return nullptr;
  return exports;
}

NAPI_MODULE(NODE_GYP_MODULE_NAME, init)

}  // namespace demo
```

The functions available and how to use them are documented in the section titled C/C++ Addons - N-API .

Addon examples

Following are some example Addons intended to help developers get started. The examples make use of the V8 APIs. Refer to the online V8 reference for help with the various V8 calls, and V8's Embedder's Guide for an explanation of several concepts used such as handles, scopes, function templates, etc.

Each of these examples using the following `binding.gyp` file:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "addon.cc" ]
    }
  ]
}
```

In cases where there is more than one `.cc` file, simply add the additional filename to the `sources` array:

```
"sources": ["addon.cc", "myexample.cc"]
```

Once the `binding.gyp` file is ready, the example Addons can be configured and built using `node-gyp` :

```
$ node-gyp configure build
```

Function arguments

Addons will typically expose objects and functions that can be accessed from JavaScript running within Node.js. When functions are invoked from JavaScript, the input arguments and return value must be mapped to and from the C/C++ code.

The following example illustrates how to read function arguments passed from JavaScript and how to return a result:

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Exception;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// This is the implementation of the "add" method
// Input arguments are passed using the
// const FunctionCallbackInfo<Value>& args struct
void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  // Check the number of arguments passed.
  if (args.Length() < 2) {
    // Throw an Error that is passed back to JavaScript
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong number of arguments")));
    return;
  }

  // Check the argument types
  if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong arguments")));
    return;
  }

  // Perform the operation
  double value = args[0]->NumberValue() + args[1]->NumberValue();
  Local<Number> num = Number::New(isolate, value);

  // Set the return value (using the passed in
  // FunctionCallbackInfo<Value>&)
  args.GetReturnValue().Set(num);
}

void Init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

Once compiled, the example Addon can be required and used from within Node.js:

```
// test.js
const addon = require('./build/Release/addon');

console.log('This should be eight:', addon.add(3, 5));
```

Callbacks

It is common practice within Addons to pass JavaScript functions to a C++ function and execute them from there. The following example illustrates how to invoke such callbacks:

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Null;
using v8::Object;
using v8::String;
using v8::Value;

void RunCallback(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Function> cb = Local<Function>::Cast(args[0]);
  const unsigned argc = 1;
  Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello world") };
  cb->Call(Null(isolate), argc, argv);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", RunCallback);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

Note that this example uses a two-argument form of `Init()` that receives the full `module` object as the second argument. This allows the Addon to completely overwrite `exports` with a single function instead of adding the function as a property of `exports`.

To test it, run the following JavaScript:

```
// test.js
const addon = require('./build/Release/addon');

addon((msg) => {
  console.log(msg);
// Prints: 'hello world'
});
```

Note that, in this example, the callback function is invoked synchronously.

Object factory

Addons can create and return new objects from within a C++ function as illustrated in the following example. An object is created and returned with a property `msg` that echoes the string passed to `createObject()` :

```
// addon.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  Local<Object> obj = Object::New(isolate);
  obj->Set(String::NewFromUtf8(isolate, "msg"), args[0]->ToString());

  args.GetReturnValue().Set(obj);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

To test it in JavaScript:

```
// test.js
const addon = require('./build/Release/addon');

const obj1 = addon('hello');
const obj2 = addon('world');
console.log(obj1.msg, obj2.msg);
// Prints: 'hello world'
```

Function factory

Another common scenario is creating JavaScript functions that wrap C++ functions and returning those back to JavaScript:

```cpp
// addon.cc
#include <node.h>

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void MyFunction(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "hello world"));
}

void CreateFunction(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, MyFunction);
  Local<Function> fn = tpl->GetFunction();

  // omit this to make it anonymous
  fn->SetName(String::NewFromUtf8(isolate, "theFunction"));

  args.GetReturnValue().Set(fn);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", CreateFunction);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

To test:

```js
// test.js
const addon = require('./build/Release/addon');

const fn = addon();
console.log(fn());
// Prints: 'hello world'
```

Wrapping C++ objects

It is also possible to wrap C++ objects/classes in a way that allows new instances to be created using the JavaScript `new` operator:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Local;
using v8::Object;

void InitAll(Local<Object> exports) {
  MyObject::Init(exports);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

}  // namespace demo
```

Then, in `myobject.h`, the wrapper class inherits from `node::ObjectWrap`:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Local<v8::Object> exports);

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

In `myobject.cc`, implement the various methods that are to be exposed. Below, the method `plusOne()` is exposed by adding it to the constructor's prototype:

```
// myobject.cc
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
```

```
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Local<Object> exports) {
  Isolate* isolate = exports->GetIsolate();

  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  constructor.Reset(isolate, tpl->GetFunction());
  exports->Set(String::NewFromUtf8(isolate, "MyObject"),
               tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Context> context = isolate->GetCurrentContext();
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Object> result =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(result);
  }
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
```

```
  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

}  // namespace demo
```

To build this example, the `myobject.cc` file must be added to the `binding.gyp`:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}
```

Test it with:

```
// test.js
const addon = require('./build/Release/addon');

const obj = new addon.MyObject(10);
console.log(obj.plusOne());
// Prints: 11
console.log(obj.plusOne());
// Prints: 12
console.log(obj.plusOne());
// Prints: 13
```

Factory of wrapped objects

Alternatively, it is possible to use a factory pattern to avoid explicitly creating object instances using the JavaScript `new` operator:

```
const obj = addon.createObject();
// instead of:
// const obj = new addon.Object();
```

First, the `createObject()` method is implemented in `addon.cc`:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  MyObject::NewInstance(args);
}

void InitAll(Local<Object> exports, Local<Object> module) {
  MyObject::Init(exports->GetIsolate());

  NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

}  // namespace demo
```

In `myobject.h`, the static method `NewInstance()` is added to handle instantiating the object. This method takes the place of using `new` in JavaScript:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Isolate* isolate);
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

The implementation in `myobject.cc` is similar to the previous example:

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
```

```
void MyObject::Init(Isolate* isolate) {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Context> context = isolate->GetCurrentContext();
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(instance);
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, constructor);
  Local<Context> context = isolate->GetCurrentContext();
  Local<Object> instance =
      cons->NewInstance(context, argc, argv).ToLocalChecked();

  args.GetReturnValue().Set(instance);
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

}  // namespace demo
```

Once again, to build this example, the `myobject.cc` file must be added to the `binding.gyp`:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}
```

Test it with:

```
// test.js
const createObject = require('./build/Release/addon');

const obj = createObject(10);
console.log(obj.plusOne());
// Prints: 11
console.log(obj.plusOne());
// Prints: 12
console.log(obj.plusOne());
// Prints: 13

const obj2 = createObject(20);
console.log(obj2.plusOne());
// Prints: 21
console.log(obj2.plusOne());
// Prints: 22
console.log(obj2.plusOne());
// Prints: 23
```

Passing wrapped objects around

In addition to wrapping and returning C++ objects, it is possible to pass wrapped objects around by unwrapping them with the Node.js helper function `node::ObjectWrap::Unwrap`. The following examples shows a function `add()` that can take two `MyObject` objects as input arguments:

```
// addon.cc
#include <node.h>
#include <node_object_wrap.h>
#include "myobject.h"

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  MyObject::NewInstance(args);
}

void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj1 = node::ObjectWrap::Unwrap<MyObject>(
      args[0]->ToObject());
  MyObject* obj2 = node::ObjectWrap::Unwrap<MyObject>(
      args[1]->ToObject());

  double sum = obj1->value() + obj2->value();
  args.GetReturnValue().Set(Number::New(isolate, sum));
}

void InitAll(Local<Object> exports) {
  MyObject::Init(exports->GetIsolate());

  NODE_SET_METHOD(exports, "createObject", CreateObject);
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

}  // namespace demo
```

In `myobject.h`, a new public method is added to allow access to private values after unwrapping the object.

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Isolate* isolate);
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);
  inline double value() const { return value_; }

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

The implementation of `myobject.cc` is similar to before:

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
  // Prepare constructor template
```

```
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Context> context = isolate->GetCurrentContext();
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(instance);
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, constructor);
  Local<Context> context = isolate->GetCurrentContext();
  Local<Object> instance =
      cons->NewInstance(context, argc, argv).ToLocalChecked();

  args.GetReturnValue().Set(instance);
}

}  // namespace demo
```

Test it with:

```
// test.js
const addon = require('./build/Release/addon');

const obj1 = addon.createObject(10);
const obj2 = addon.createObject(20);
const result = addon.add(obj1, obj2);

console.log(result);
// Prints: 30
```

AtExit hooks

An "AtExit" hook is a function that is invoked after the Node.js event loop has ended but before the JavaScript VM is terminated and Node.js shuts down. "AtExit" hooks are registered using the `node::AtExit` API.

void AtExit(callback, args)

- `callback` <void (*)(void*)> A pointer to the function to call at exit.
- `args` <void*> A pointer to pass to the callback at exit.

Registers exit hooks that run after the event loop has ended but before the VM is killed.

AtExit takes two parameters: a pointer to a callback function to run at exit, and a pointer to untyped context data to be passed to that callback.

Callbacks are run in last-in first-out order.

The following `addon.cc` implements AtExit:

```
// addon.cc
#include <assert.h>
#include <stdlib.h>
#include <node.h>

namespace demo {

using node::AtExit;
using v8::HandleScope;
using v8::Isolate;
using v8::Local;
using v8::Object;

static char cookie[] = "yum yum";
static int at_exit_cb1_called = 0;
static int at_exit_cb2_called = 0;

static void at_exit_cb1(void* arg) {
  Isolate* isolate = static_cast<Isolate*>(arg);
  HandleScope scope(isolate);
  Local<Object> obj = Object::New(isolate);
  assert(!obj.IsEmpty());  // assert VM is still alive
  assert(obj->IsObject());
  at_exit_cb1_called++;
}

static void at_exit_cb2(void* arg) {
  assert(arg == static_cast<void*>(cookie));
  at_exit_cb2_called++;
}

static void sanity_check(void*) {
  assert(at_exit_cb1_called == 1);
  assert(at_exit_cb2_called == 2);
}

void init(Local<Object> exports) {
  AtExit(at_exit_cb2, cookie);
  AtExit(at_exit_cb2, cookie);
  AtExit(at_exit_cb1, exports->GetIsolate());
  AtExit(sanity_check);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, init)

}  // namespace demo
```

Test in JavaScript by running:

```
// test.js
require('./build/Release/addon');
```

N-API

Stability: 1 - Experimental

N-API (pronounced N as in the letter, followed by API) is an API for building native Addons. It is independent from the underlying JavaScript runtime (ex V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate Addons from changes in the underlying JavaScript engine and allow modules compiled for one version to run on later versions of Node.js without recompilation.

Addons are built/packaged with the same approach/tools outlined in the section titled C++ Addons. The only difference is the set of APIs that are used by the native code. Instead of using the V8 or Native Abstractions for Node.js APIs, the functions available in the N-API are used.

APIs exposed by N-API are generally used to create and manipulate JavaScript values. Concepts and operations generally map to ideas specified in the ECMA262 Language Specification. The APIs have the following properties:

- All N-API calls return a status code of type `napi_status`. This status indicates whether the API call succeeded or failed.
- The API's return value is passed via an out parameter.
- All JavaScript values are abstracted behind an opaque type named `napi_value`.
- In case of an error status code, additional information can be obtained using `napi_get_last_error_info`. More information can be found in the error handling section Error Handling.

The documentation for N-API is structured as follows:

- Basic N-API Data Types
- Error Handling
- Object Lifetime Management
- Module Registration
- Working with JavaScript Values
- Working with JavaScript Values - Abstract Operations
- Working with JavaScript Properties
- Working with JavaScript Functions
- Object Wrap
- Simple Asynchronous Operations
- Custom Asynchronous Operations
- Promises
- Script Execution

The N-API is a C API that ensures ABI stability across Node.js versions and different compiler levels. However, we also understand that a C++ API can be easier to use in many cases. To support these cases we expect there to be one or more C++ wrapper modules that provide an inlineable C++ API. Binaries built with these wrapper modules will depend on the symbols for the N-API C based functions exported by Node.js. These wrappers are not part of N-API, nor will they be maintained as part of Node.js. One such example is: node-api.

In order to use the N-API functions, include the file node_api.h which is located in the src directory in the node development tree:

```
#include <node_api.h>
```

Basic N-API Data Types

N-API exposes the following fundamental datatypes as abstractions that are consumed by the various APIs. These APIs should be treated as opaque, introspectable only with other N-API calls.

napi_status

Integral status code indicating the success or failure of a N-API call. Currently, the following status codes are supported.

```
typedef enum {
  napi_ok,
  napi_invalid_arg,
  napi_object_expected,
  napi_string_expected,
  napi_name_expected,
  napi_function_expected,
  napi_number_expected,
  napi_boolean_expected,
  napi_array_expected,
  napi_generic_failure,
  napi_pending_exception,
  napi_cancelled,
  napi_status_last
} napi_status;
```

If additional information is required upon an API returning a failed status, it can be obtained by calling `napi_get_last_error_info`.

napi_extended_error_info

```
typedef struct {
  const char* error_message;
  void* engine_reserved;
  uint32_t engine_error_code;
  napi_status error_code;
} napi_extended_error_info;
```

- `error_message`: UTF8-encoded string containing a VM-neutral description of the error.
- `engine_reserved`: Reserved for VM-specific error details. This is currently not implemented for any VM.
- `engine_error_code`: VM-specific error code. This is currently not implemented for any VM.
- `error_code`: The N-API status code that originated with the last error.

See the Error Handling section for additional information.

napi_env

`napi_env` is used to represent a context that the underlying N-API implementation can use to persist VM-specific state. This structure is passed to native functions when they're invoked, and it must be passed back when making N-API calls. Specifically, the same `napi_env` that was passed in when the initial native function was called must be passed to any subsequent nested N-API calls. Caching the `napi_env` for the purpose of general reuse is not allowed.

napi_value

This is an opaque pointer that is used to represent a JavaScript value.

N-API Memory Management types

napi_handle_scope

This is an abstraction used to control and modify the lifetime of objects created within a particular scope. In general, N-API values are created within the context of a handle scope. When a native method is called from JavaScript, a default handle scope will exist. If the user does not explicitly create a new handle scope, N-API values will be created in the default handle scope. For any invocations of code outside the execution of a native method (for instance, during a libuv callback invocation), the module is required to create a scope before invoking any functions that can result in the creation of JavaScript values.

Handle scopes are created using `napi_open_handle_scope` and are destroyed using `napi_close_handle_scope`. Closing the scope can indicate to

the GC that all `napi_value` s created during the lifetime of the handle scope are no longer referenced from the current stack frame.

For more details, review the Object Lifetime Management .

napi_escapable_handle_scope

Escapable handle scopes are a special type of handle scope to return values created within a particular handle scope to a parent scope.

napi_ref

This is the abstraction to use to reference a `napi_value` . This allows for users to manage the lifetimes of JavaScript values, including defining their minimum lifetimes explicitly.

For more details, review the Object Lifetime Management .

N-API Callback types

napi_callback_info

Opaque datatype that is passed to a callback function. It can be used for getting additional information about the context in which the callback was invoked.

napi_callback

Function pointer type for user-provided native functions which are to be exposed to JavaScript via N-API. Callback functions should satisfy the following signature:

```
typedef napi_value (*napi_callback)(napi_env, napi_callback_info);
```

napi_finalize

Function pointer type for add-on provided functions that allow the user to be notified when externally-owned data is ready to be cleaned up because the object with which it was associated with, has been garbage-collected. The user must provide a function satisfying the following signature which would get called upon the object's collection. Currently, `napi_finalize` can be used for finding out when objects that have external data are collected.

```
typedef void (*napi_finalize)(napi_env env,
                              void* finalize_data,
                              void* finalize_hint);
```

napi_async_execute_callback

Function pointer used with functions that support asynchronous operations. Callback functions must statisfy the following signature:

```
typedef void (*napi_async_execute_callback)(napi_env env, void* data);
```

napi_async_complete_callback

Function pointer used with functions that support asynchronous operations. Callback functions must statisfy the following signature:

```
typedef void (*napi_async_complete_callback)(napi_env env,
                                             napi_status status,
                                             void* data);
```

Error Handling

N-API uses both return values and JavaScript exceptions for error handling. The following sections explain the approach for each case.

Return values

All of the N-API functions share the same error handling pattern. The return type of all API functions is `napi_status`.

The return value will be `napi_ok` if the request was successful and no uncaught JavaScript exception was thrown. If an error occurred AND an exception was thrown, the `napi_status` value for the error will be returned. If an exception was thrown, and no error occurred, `napi_pending_exception` will be returned.

In cases where a return value other than `napi_ok` or `napi_pending_exception` is returned, `napi_is_exception_pending` must be called to check if an exception is pending. See the section on exceptions for more details.

The full set of possible napi_status values is defined in `napi_api_types.h`.

The `napi_status` return value provides a VM-independent representation of the error which occurred. In some cases it is useful to be able to get more detailed information, including a string representing the error as well as VM (engine)-specific information.

In order to retrieve this information `napi_get_last_error_info` is provided which returns a `napi_extended_error_info` structure. The format of the `napi_extended_error_info` structure is as follows:

```
typedef struct napi_extended_error_info {
  const char* error_message;
  void* engine_reserved;
  uint32_t engine_error_code;
  napi_status error_code;
};
```

- `error_message`: Textual representation of the error that occurred.
- `engine_reserved`: Opaque handle reserved for engine use only.
- `engine_error_code`: VM specific error code.
- `error_code`: n-api status code for the last error.

`napi_get_last_error_info` returns the information for the last N-API call that was made.

Do not rely on the content or format of any of the extended information as it is not subject to SemVer and may change at any time. It is intended only for logging purposes.

napi_get_last_error_info

Added in: v8.0.0

```
napi_status
napi_get_last_error_info(napi_env env,
                         const napi_extended_error_info** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: The `napi_extended_error_info` structure with more information about the error.

Returns `napi_ok` if the API succeeded.

This API retrieves a `napi_extended_error_info` structure with information about the last error that occurred.

The content of the `napi_extended_error_info` returned is only valid up until an n-api function is called on the same `env`.

Do not rely on the content or format of any of the extended information as it is not subject to SemVer and may change at any time. It is intended only for logging purposes.

This API can be called even if there is a pending JavaScript exception.

Exceptions

Any N-API function call may result in a pending JavaScript exception. This is obviously the case for any function that may cause the execution of JavaScript, but N-API specifies that an exception may be pending on return from any of the API functions.

If the `napi_status` returned by a function is `napi_ok` then no exception is pending and no additional action is required. If the `napi_status` returned is anything other than `napi_ok` or `napi_pending_exception`, in order to try to recover and continue instead of simply returning immediately, `napi_is_exception_pending` must be called in order to determine if an exception is pending or not.

When an exception is pending one of two approaches can be employed.

The first approach is to do any appropriate cleanup and then return so that execution will return to JavaScript. As part of the transition back to JavaScript the exception will be thrown at the point in the JavaScript code where the native method was invoked. The behavior of most N-API calls is unspecified while an exception is pending, and many will simply return `napi_pending_exception`, so it is important to do as little as possible and then return to JavaScript where the exception can be handled.

The second approach is to try to handle the exception. There will be cases where the native code can catch the exception, take the appropriate action, and then continue. This is only recommended in specific cases where it is known that the exception can be safely handled. In these cases `napi_get_and_clear_last_exception` can be used to get and clear the exception. On success, result will contain the handle to the last JavaScript Object thrown. If it is determined, after retrieving the exception, the exception cannot be handled after all it can be re-thrown it with `napi_throw` where error is the JavaScript Error object to be thrown.

The following utility functions are also available in case native code needs to throw an exception or determine if a `napi_value` is an instance of a JavaScript `Error` object: `napi_throw_error`, `napi_throw_type_error`, `napi_throw_range_error` and `napi_is_error`.

The following utility functions are also available in case native code needs to create an Error object: `napi_create_error`, `napi_create_type_error`, and `napi_create_range_error`. where result is the napi_value that refers to the newly created JavaScript Error object.

The Node.js project is adding error codes to all of the errors generated internally. The goal is for applications to use these error codes for all error checking. The associated error messages will remain, but will only be meant to be used for logging and display with the expectation that the message can change without SemVer applying. In order to support this model with N-API, both in internal functionality and for module specific functionality (as its good practice), the `throw_` and `create_` functions take an optional code parameter which is the string for the code to be added to the error object. If the optional parameter is NULL then no code will be associated with the error. If a code is provided, the name associated with the error is also updated to be:

```
originalName [code]
```

where originalName is the original name associated with the error and code is the code that was provided. For example if the code is 'ERR_ERROR_1' and a TypeError is being created the name will be:

```
TypeError [ERR_ERROR_1]
```

napi_throw

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_throw(napi_env env, napi_value error);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] error`: The `napi_value` for the Error to be thrown.

Returns `napi_ok` if the API succeeded.

This API throws the JavaScript Error provided.

napi_throw_error

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_throw_error(napi_env env,
                                         const char* code,
                                         const char* msg);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional error code to be set on the error.
- `[in] msg` : C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript Error with the text provided.

napi_throw_type_error

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_throw_type_error(napi_env env,
                                              const char* code,
                                              const char* msg);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional error code to be set on the error.
- `[in] msg` : C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript TypeError with the text provided.

napi_throw_range_error

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_throw_range_error(napi_env env,
                                               const char* code,
                                               const char* msg);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional error code to be set on the error.
- `[in] msg` : C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript RangeError with the text provided.

napi_is_error

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_is_error(napi_env env,
                                      napi_value value,
                                      bool* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] msg` : The `napi_value` to be checked.
- `[out] result` : Boolean value that is set to true if `napi_value` represents an error, false otherwise.

Returns `napi_ok` if the API succeeded.

This API queries a `napi_value` to check if it represents an error object.

napi_create_error

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_create_error(napi_env env,
                                          napi_value code,
                                          napi_value msg,
                                          napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional `napi_value` with the string for the error code to

```
be associated with the error.
```

- `[in] msg` : napi_value that references a JavaScript String to be used as the message for the Error.
- `[out] result` : `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript Error with the text provided.

napi_create_type_error

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_create_type_error(napi_env env,
                                               napi_value code,
                                               napi_value msg,
                                               napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional `napi_value` with the string for the error code to

```
be associated with the error.
```

- `[in] msg` : napi_value that references a JavaScript String to be used as the message for the Error.
- `[out] result` : `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript TypeError with the text provided.

napi_create_range_error

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_create_range_error(napi_env env,
                                                napi_value code,
                                                const char* msg,
                                                napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional `napi_value` with the string for the error code to

```
be associated with the error.
```

- `[in] msg` : napi_value that references a JavaScript String to be used as the message for the Error.
- `[out] result` : `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript RangeError with the text provided.

napi_get_and_clear_last_exception

Added in: v8.0.0

```
napi_status napi_get_and_clear_last_exception(napi_env env,
                                              napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : The exception if one is pending, NULL otherwise.

Returns `napi_ok` if the API succeeded.

This API returns true if an exception is pending.

This API can be called even if there is a pending JavaScript exception.

napi_is_exception_pending

Added in: v8.0.0

```
napi_status napi_is_exception_pending(napi_env env, bool* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : Boolean value that is set to true if an exception is pending.

Returns `napi_ok` if the API succeeded.

This API returns true if an exception is pending.

This API can be called even if there is a pending JavaScript exception.

napi_fatal_exception

Added in: v9.10.0

```
napi_status napi_fatal_exception(napi_env env, napi_value err);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] err` : The error you want to pass to `uncaughtException`.

Trigger an `uncaughtException` in JavaScript. Useful if an async callback throws an exception with no way to recover.

Fatal Errors

In the event of an unrecoverable error in a native module, a fatal error can be thrown to immediately terminate the process.

napi_fatal_error

Added in: v8.2.0

```
NAPI_NO_RETURN void napi_fatal_error(const char* location,
                                      size_t location_len,
                                      const char* message,
                                      size_t message_len);
```

- `[in] location` : Optional location at which the error occurred.
- `[in] location_len` : The length of the location in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] message` : The message associated with the error.
- `[in] message_len` : The length of the message in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.

The function call does not return, the process will be terminated.

This API can be called even if there is a pending JavaScript exception.

Object Lifetime management

As N-API calls are made, handles to objects in the heap for the underlying VM may be returned as `napi_values` . These handles must hold the objects 'live' until they are no longer required by the native code, otherwise the objects could be collected before the native code was finished using them.

As object handles are returned they are associated with a 'scope'. The lifespan for the default scope is tied to the lifespan of the native method call. The result is that, by default, handles remain valid and the objects associated with these handles will be held live for the lifespan of the native method call.

In many cases, however, it is necessary that the handles remain valid for either a shorter or longer lifespan than that of the native method. The sections which follow describe the N-API functions than can be used to change the handle lifespan from the default.

Making handle lifespan shorter than that of the native method

It is often necessary to make the lifespan of handles shorter than the lifespan of a native method. For example, consider a native method that has a loop which iterates through the elements in a large array:

```
for (int i = 0; i < 1000000; i++) {
  napi_value result;
  napi_status status = napi_get_element(e, object, i, &result);
  if (status != napi_ok) {
    break;
  }
  // do something with element
}
```

This would result in a large number of handles being created, consuming substantial resources. In addition, even though the native code could only use the most recent handle, all of the associated objects would also be kept alive since they all share the same scope.

To handle this case, N-API provides the ability to establish a new 'scope' to which newly created handles will be associated. Once those handles are no longer required, the scope can be 'closed' and any handles associated with the scope are invalidated. The methods available to open/close scopes are `napi_open_handle_scope` and `napi_close_handle_scope` .

N-API only supports a single nested hierarchy of scopes. There is only one active scope at any time, and all new handles will be associated with that scope while it is active. Scopes must be closed in the reverse order from which they are opened. In addition, all scopes created within a native method must be closed before returning from that method.

Taking the earlier example, adding calls to `napi_open_handle_scope` and `napi_close_handle_scope` would ensure that at most a single handle is valid throughout the execution of the loop:

```
for (int i = 0; i < 1000000; i++) {
  napi_handle_scope scope;
  napi_status status = napi_open_handle_scope(env, &scope);
  if (status != napi_ok) {
    break;
  }
  napi_value result;
  status = napi_get_element(e, object, i, &result);
  if (status != napi_ok) {
    break;
  }
  // do something with element
  status = napi_close_handle_scope(env, scope);
  if (status != napi_ok) {
    break;
  }
}
```

When nesting scopes, there are cases where a handle from an inner scope needs to live beyond the lifespan of that scope. N-API supports an 'escapable scope' in order to support this case. An escapable scope allows one handle to be 'promoted' so that it 'escapes' the current scope and the lifespan of the handle changes from the current scope to that of the outer scope.

The methods available to open/close escapable scopes are `napi_open_escapable_handle_scope` and `napi_close_escapable_handle_scope`.

The request to promote a handle is made through `napi_escape_handle` which can only be called once.

napi_open_handle_scope

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_open_handle_scope(napi_env env,
                                               napi_handle_scope* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: `napi_value` representing the new scope.

Returns `napi_ok` if the API succeeded.

This API open a new scope.

napi_close_handle_scope

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_close_handle_scope(napi_env env,
                                                napi_handle_scope scope);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] scope`: `napi_value` representing the scope to be closed.

Returns `napi_ok` if the API succeeded.

This API closes the scope passed in. Scopes must be closed in the reverse order from which they were created.

This API can be called even if there is a pending JavaScript exception.

napi_open_escapable_handle_scope

Added in: v8.0.0

```
NODE_EXTERN napi_status
    napi_open_escapable_handle_scope(napi_env env,
                                      napi_handle_scope* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing the new scope.

Returns `napi_ok` if the API succeeded.

This API open a new scope from which one object can be promoted to the outer scope.

napi_close_escapable_handle_scope

Added in: v8.0.0

```
NODE_EXTERN napi_status
    napi_close_escapable_handle_scope(napi_env env,
                                       napi_handle_scope scope);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] scope` : `napi_value` representing the scope to be closed.

Returns `napi_ok` if the API succeeded.

This API closes the scope passed in. Scopes must be closed in the reverse order from which they were created.

This API can be called even if there is a pending JavaScript exception.

napi_escape_handle

Added in: v8.0.0

```
napi_status napi_escape_handle(napi_env env,
                               napi_escapable_handle_scope scope,
                               napi_value escapee,
                               napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] scope` : `napi_value` representing the current scope.
- `[in] escapee` : `napi_value` representing the JavaScript Object to be escaped.
- `[out] result` : `napi_value` representing the handle to the escaped Object in the outer scope.

Returns `napi_ok` if the API succeeded.

This API promotes the handle to the JavaScript object so that it is valid for the lifetime of the outer scope. It can only be called once per scope. If it is called more than once an error will be returned.

This API can be called even if there is a pending JavaScript exception.

References to objects with a lifespan longer than that of the native method

In some cases an addon will need to be able to create and reference objects with a lifespan longer than that of a single native method invocation. For example, to create a constructor and later use that constructor in a request to creates instances, it must be possible to reference the constructor object across many different instance creation requests. This would not be possible with a normal handle returned as a `napi_value` as described in the earlier section. The lifespan of a normal handle is managed by scopes and all scopes must be closed before the end of a native method.

N-API provides methods to create persistent references to an object. Each persistent reference has an associated count with a value of 0 or higher. The count determines if the reference will keep the corresponding object live. References with a count of 0 do not prevent the object

from being collected and are often called 'weak' references. Any count greater than 0 will prevent the object from being collected.

References can be created with an initial reference count. The count can then be modified through `napi_reference_ref` and `napi_reference_unref`. If an object is collected while the count for a reference is 0, all subsequent calls to get the object associated with the reference `napi_get_reference_value` will return NULL for the returned `napi_value`. An attempt to call `napi_reference_ref` for a reference whose object has been collected will result in an error.

References must be deleted once they are no longer required by the addon. When a reference is deleted it will no longer prevent the corresponding object from being collected. Failure to delete a persistent reference will result in a 'memory leak' with both the native memory for the persistent reference and the corresponding object on the heap being retained forever.

There can be multiple persistent references created which refer to the same object, each of which will either keep the object live or not based on its individual count.

napi_create_reference

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_create_reference(napi_env env,
                                              napi_value value,
                                              int initial_refcount,
                                              napi_ref* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: `napi_value` representing the Object to which we want a reference.
- `[in] initial_refcount`: Initial reference count for the new reference.
- `[out] result`: `napi_ref` pointing to the new reference.

Returns `napi_ok` if the API succeeded.

This API create a new reference with the specified reference count to the Object passed in.

napi_delete_reference

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_delete_reference(napi_env env, napi_ref ref);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] ref`: `napi_ref` to be deleted.

Returns `napi_ok` if the API succeeded.

This API deletes the reference passed in.

This API can be called even if there is a pending JavaScript exception.

napi_reference_ref

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_reference_ref(napi_env env,
                                           napi_ref ref,
                                           int* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] ref`: `napi_ref` for which the reference count will be incremented.
- `[out] result`: The new reference count.

Returns `napi_ok` if the API succeeded.

This API increments the reference count for the reference passed in and returns the resulting reference count.

napi_reference_unref

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_reference_unref(napi_env env,
                                             napi_ref ref,
                                             int* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which the reference count will be decremented.
- `[out] result` : The new reference count.

Returns `napi_ok` if the API succeeded.

This API decrements the reference count for the reference passed in and returns the resulting reference count.

napi_get_reference_value

Added in: v8.0.0

```
NODE_EXTERN napi_status napi_get_reference_value(napi_env env,
                                                 napi_ref ref,
                                                 napi_value* result);
```

the `napi_value passed` in or out of these methods is a handle to the object to which the reference is related.

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which we requesting the corresponding Object.
- `[out] result` : The `napi_value` for the Object referenced by the `napi_ref`.

Returns `napi_ok` if the API succeeded.

If still valid, this API returns the `napi_value` representing the JavaScript Object associated with the `napi_ref`. Otherwise, result will be NULL.

Module registration

N-API modules are registered in a manner similar to other modules except that instead of using the `NODE_MODULE` macro the following is used:

```
NAPI_MODULE(NODE_GYP_MODULE_NAME, Init)
```

The next difference is the signature for the `Init` method. For a N-API module it is as follows:

```
napi_value Init(napi_env env, napi_value exports);
```

The return value from `Init` is treated as the `exports` object for the module. The `Init` method is passed an empty object via the `exports` parameter as a convenience. If `Init` returns NULL, the parameter passed as `exports` is exported by the module. N-API modules cannot modify the `module` object but can specify anything as the `exports` property of the module.

For example, to add the method `hello` as a function so that it can be called as a method provided by the addon:

```
napi_value Init(napi_env env, napi_value exports) {
  napi_status status;
  napi_property_descriptor desc =
    {"hello", Method, 0, 0, 0, napi_default, 0};
  if (status != napi_ok) return NULL;
  status = napi_define_properties(env, exports, 1, &desc);
  if (status != napi_ok) return NULL;
  return exports;
}
```

For example, to set a function to be returned by the `require()` for the addon:

```
napi_value Init(napi_env env, napi_value exports) {
  napi_value method;
  napi_status status;
  status = napi_create_function(env, "exports", NAPI_AUTO_LENGTH, Method, NULL, &method);
  if (status != napi_ok) return NULL;
  return method;
}
```

For example, to define a class so that new instances can be created (often used with Object Wrap):

```
// NOTE: partial example, not all referenced code is included
napi_value Init(napi_env env, napi_value exports) {
  napi_status status;
  napi_property_descriptor properties[] = {
    { "value", NULL, GetValue, SetValue, 0, napi_default, 0 },
    DECLARE_NAPI_METHOD("plusOne", PlusOne),
    DECLARE_NAPI_METHOD("multiply", Multiply),
  };

  napi_value cons;
  status =
      napi_define_class(env, "MyObject", New, NULL, 3, properties, &cons);
  if (status != napi_ok) return NULL;

  status = napi_create_reference(env, cons, 1, &constructor);
  if (status != napi_ok) return NULL;

  status = napi_set_named_property(env, exports, "MyObject", cons);
  if (status != napi_ok) return NULL;

  return exports;
}
```

For more details on setting properties on objects, see the section on Working with JavaScript Properties .

For more details on building addon modules in general, refer to the existing API

Working with JavaScript Values

N-API exposes a set of APIs to create all types of JavaScript values. Some of these types are documented under Section 6 of the ECMAScript Language Specification .

Fundamentally, these APIs are used to do one of the following:

1. Create a new JavaScript object
2. Convert from a primitive C type to an N-API value
3. Convert from N-API value to a primitive C type
4. Get global instances including `undefined` and `null`

N-API values are represented by the type `napi_value`. Any N-API call that requires a JavaScript value takes in a `napi_value`. In some cases, the API does check the type of the `napi_value` up-front. However, for better performance, it's better for the caller to make sure that the `napi_value` in question is of the JavaScript type expected by the API.

Enum types

napi_valuetype

```
typedef enum {
  // ES6 types (corresponds to typeof)
  napi_undefined,
  napi_null,
  napi_boolean,
  napi_number,
  napi_string,
  napi_symbol,
  napi_object,
  napi_function,
  napi_external,
} napi_valuetype;
```

Describes the type of a `napi_value`. This generally corresponds to the types described in Section 6.1 of the ECMAScript Language Specification. In addition to types in that section, `napi_valuetype` can also represent Functions and Objects with external data.

napi_typedarray_type

```
typedef enum {
  napi_int8_array,
  napi_uint8_array,
  napi_uint8_clamped_array,
  napi_int16_array,
  napi_uint16_array,
  napi_int32_array,
  napi_uint32_array,
  napi_float32_array,
  napi_float64_array,
} napi_typedarray_type;
```

This represents the underlying binary scalar datatype of the TypedArray. Elements of this enum correspond to Section 22.2 of the ECMAScript Language Specification.

Object Creation Functions

napi_create_array

Added in: v8.0.0

```
napi_status napi_create_array(napi_env env, napi_value* result)
```

- `[in] env`: The environment that the N-API call is invoked under.
- `[out] result`: A `napi_value` representing a JavaScript Array.

Returns `napi_ok` if the API succeeded.

This API returns an N-API value corresponding to a JavaScript Array type. JavaScript arrays are described in Section 22.1 of the ECMAScript Language Specification.

napi_create_array_with_length

Added in: v8.0.0

```
napi_status napi_create_array_with_length(napi_env env,
                                          size_t length,
                                          napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] length` : The initial length of the Array.
- `[out] result` : A `napi_value` representing a JavaScript Array.

Returns `napi_ok` if the API succeeded.

This API returns an N-API value corresponding to a JavaScript Array type. The Array's length property is set to the passed-in length parameter. However, the underlying buffer is not guaranteed to be pre-allocated by the VM when the array is created - that behavior is left to the underlying VM implementation. If the buffer must be a contiguous block of memory that can be directly read and/or written via C, consider using `napi_create_external_arraybuffer` .

JavaScript arrays are described in Section 22.1 of the ECMAScript Language Specification.

napi_create_arraybuffer

Added in: v8.0.0

```
napi_status napi_create_arraybuffer(napi_env env,
                                    size_t byte_length,
                                    void** data,
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] length` : The length in bytes of the array buffer to create.
- `[out] data` : Pointer to the underlying byte buffer of the ArrayBuffer.
- `[out] result` : A `napi_value` representing a JavaScript ArrayBuffer.

Returns `napi_ok` if the API succeeded.

This API returns an N-API value corresponding to a JavaScript ArrayBuffer. ArrayBuffers are used to represent fixed-length binary data buffers. They are normally used as a backing-buffer for TypedArray objects. The ArrayBuffer allocated will have an underlying byte buffer whose size is determined by the `length` parameter that's passed in. The underlying buffer is optionally returned back to the caller in case the caller wants to directly manipulate the buffer. This buffer can only be written to directly from native code. To write to this buffer from JavaScript, a typed array or DataView object would need to be created.

JavaScript ArrayBuffer objects are described in Section 24.1 of the ECMAScript Language Specification.

napi_create_buffer

Added in: v8.0.0

```
napi_status napi_create_buffer(napi_env env,
                               size_t size,
                               void** data,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] size` : Size in bytes of the underlying buffer.
- `[out] data` : Raw pointer to the underlying buffer.
- `[out] result` : A `napi_value` representing a `node::Buffer`.

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object. While this is still a fully-supported data structure, in most cases using a TypedArray will suffice.

napi_create_buffer_copy

Added in: v8.0.0

```
napi_status napi_create_buffer_copy(napi_env env,
                                    size_t length,
                                    const void* data,
                                    void** result_data,
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] size` : Size in bytes of the input buffer (should be the same as the size of the new buffer).
- `[in] data` : Raw pointer to the underlying buffer to copy from.
- `[out] result_data` : Pointer to the new Buffer's underlying data buffer.
- `[out] result` : A `napi_value` representing a `node::Buffer`.

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object and initializes it with data copied from the passed-in buffer. While this is still a fully-supported data structure, in most cases using a TypedArray will suffice.

napi_create_external

Added in: v8.0.0

```
napi_status napi_create_external(napi_env env,
                                 void* data,
                                 napi_finalize finalize_cb,
                                 void* finalize_hint,
                                 napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] data` : Raw pointer to the external data.
- `[in] finalize_cb` : Optional callback to call when the external value is being collected.
- `[in] finalize_hint` : Optional hint to pass to the finalize callback during collection.
- `[out] result` : A `napi_value` representing an external value.

Returns `napi_ok` if the API succeeded.

This API allocates a JavaScript value with external data attached to it. This is used to pass external data through JavaScript code, so it can be retrieved later by native code. The API allows the caller to pass in a finalize callback, in case the underlying native resource needs to be cleaned up when the external JavaScript value gets collected.

The created value is not an object, and therefore does not support additional properties. It is considered a distinct value type: calling `napi_typeof()` with an external value yields `napi_external`.

napi_create_external_arraybuffer

Added in: v8.0.0

```
napi_status
napi_create_external_arraybuffer(napi_env env,
                                 void* external_data,
                                 size_t byte_length,
                                 napi_finalize finalize_cb,
                                 void* finalize_hint,
                                 napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] external_data` : Pointer to the underlying byte buffer of the ArrayBuffer.
- `[in] byte_length` : The length in bytes of the underlying buffer.
- `[in] finalize_cb` : Optional callback to call when the ArrayBuffer is being collected.
- `[in] finalize_hint` : Optional hint to pass to the finalize callback during collection.
- `[out] result` : A `napi_value` representing a JavaScript ArrayBuffer.

Returns `napi_ok` if the API succeeded.

This API returns an N-API value corresponding to a JavaScript ArrayBuffer. The underlying byte buffer of the ArrayBuffer is externally allocated and managed. The caller must ensure that the byte buffer remains valid until the finalize callback is called.

JavaScript ArrayBuffers are described in Section 24.1 of the ECMAScript Language Specification.

napi_create_external_buffer

Added in: v8.0.0

```
napi_status napi_create_external_buffer(napi_env env,
                                        size_t length,
                                        void* data,
                                        napi_finalize finalize_cb,
                                        void* finalize_hint,
                                        napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] length` : Size in bytes of the input buffer (should be the same as the size of the new buffer).
- `[in] data` : Raw pointer to the underlying buffer to copy from.
- `[in] finalize_cb` : Optional callback to call when the ArrayBuffer is being collected.
- `[in] finalize_hint` : Optional hint to pass to the finalize callback during collection.
- `[out] result` : A `napi_value` representing a `node::Buffer` .

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object and initializes it with data backed by the passed in buffer. While this is still a fully-supported data structure, in most cases using a TypedArray will suffice.

For Node.js >=4 `Buffers` are Uint8Arrays.

napi_create_function

Added in: v8.0.0

```
napi_status napi_create_function(napi_env env,
                                 const char* utf8name,
                                 size_t length,
                                 napi_callback cb,
                                 void* data,
                                 napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] utf8name` : A string representing the name of the function encoded as UTF8.
- `[in] length` : The length of the utf8name in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] cb` : A function pointer to the native function to be invoked when the created function is invoked from JavaScript.
- `[in] data` : Optional arbitrary context data to be passed into the native function when it is invoked.
- `[out] result` : A `napi_value` representing a JavaScript function.

Returns `napi_ok` if the API succeeded.

This API returns an N-API value corresponding to a JavaScript Function object. It's used to wrap native functions so that they can be invoked from JavaScript.

JavaScript Functions are described in Section 19.2 of the ECMAScript Language Specification.

napi_create_object

Added in: v8.0.0

```
napi_status napi_create_object(napi_env env, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : A `napi_value` representing a JavaScript Object.

Returns `napi_ok` if the API succeeded.

This API allocates a default JavaScript Object. It is the equivalent of doing `new Object()` in JavaScript.

The JavaScript Object type is described in Section 6.1.7 of the ECMAScript Language Specification.

napi_create_symbol

Added in: v8.0.0

```
napi_status napi_create_symbol(napi_env env,
                               napi_value description,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] description` : Optional napi_value which refers to a JavaScript String to be set as the description for the symbol.
- `[out] result` : A `napi_value` representing a JavaScript Symbol.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript Symbol object from a UTF8-encoded C string

The JavaScript Symbol type is described in Section 19.4 of the ECMAScript Language Specification.

napi_create_typedarray

Added in: v8.0.0

```
napi_status napi_create_typedarray(napi_env env,
                                   napi_typedarray_type type,
                                   size_t length,
                                   napi_value arraybuffer,
                                   size_t byte_offset,
                                   napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] type` : Scalar datatype of the elements within the TypedArray.
- `[in] length` : Number of elements in the TypedArray.
- `[in] arraybuffer` : ArrayBuffer underlying the typed array.
- `[in] byte_offset` : The byte offset within the ArrayBuffer from which to start projecting the TypedArray.
- `[out] result` : A `napi_value` representing a JavaScript TypedArray.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript TypedArray object over an existing ArrayBuffer. TypedArray objects provide an array-like view over an underlying data buffer where each element has the same underlying binary scalar datatype.

It's required that (length * size_of_element) + byte_offset should be <= the size in bytes of the array passed in. If not, a RangeError exception is raised.

JavaScript TypedArray Objects are described in Section 22.2 of the ECMAScript Language Specification.

napi_create_dataview

Added in: v8.3.0

```
napi_status napi_create_dataview(napi_env env,
                                 size_t byte_length,
                                 napi_value arraybuffer,
                                 size_t byte_offset,
                                 napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] length` : Number of elements in the DataView.
- `[in] arraybuffer` : ArrayBuffer underlying the DataView.
- `[in] byte_offset` : The byte offset within the ArrayBuffer from which to start projecting the DataView.
- `[out] result` : A `napi_value` representing a JavaScript DataView.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript DataView object over an existing ArrayBuffer. DataView objects provide an array-like view over an underlying data buffer, but one which allows items of different size and type in the ArrayBuffer.

It is required that `byte_length + byte_offset` is less than or equal to the size in bytes of the array passed in. If not, a RangeError exception is raised.

JavaScript DataView Objects are described in Section 24.3 of the ECMAScript Language Specification.

Functions to convert from C types to N-API

napi_create_int32

Added in: v8.4.0

```
napi_status napi_create_int32(napi_env env, int32_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript Number.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `int32_t` type to the JavaScript Number type.

The JavaScript Number type is described in Section 6.1.6 of the ECMAScript Language Specification.

napi_create_uint32

Added in: v8.4.0

```
napi_status napi_create_uint32(napi_env env, uint32_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Unsigned integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript Number.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `uint32_t` type to the JavaScript Number type.

The JavaScript Number type is described in Section 6.1.6 of the ECMAScript Language Specification.

napi_create_int64

Added in: v8.4.0

```
napi_status napi_create_int64(napi_env env, int64_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript Number.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `int64_t` type to the JavaScript Number type.

The JavaScript Number type is described in Section 6.1.6 of the ECMAScript Language Specification. Note the complete range of `int64_t` cannot be represented with full precision in JavaScript. Integer values outside the range of `Number.MIN_SAFE_INTEGER` -(2^53 - 1) - `Number.MAX_SAFE_INTEGER` (2^53 - 1) will lose precision.

napi_create_double

Added in: v8.4.0

```
napi_status napi_create_double(napi_env env, double value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Double-precision value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript Number.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `double` type to the JavaScript Number type.

The JavaScript Number type is described in Section 6.1.6 of the ECMAScript Language Specification.

napi_create_string_latin1

Added in: v8.0.0

```
napi_status napi_create_string_latin1(napi_env env,
                                      const char* str,
                                      size_t length,
                                      napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] str` : Character buffer representing a ISO-8859-1-encoded string.
- `[in] length` : The length of the string in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result` : A `napi_value` representing a JavaScript String.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript String object from a ISO-8859-1-encoded C string.

The JavaScript String type is described in Section 6.1.4 of the ECMAScript Language Specification.

napi_create_string_utf16

Added in: v8.0.0

```
napi_status napi_create_string_utf16(napi_env env,
                                     const char16_t* str,
                                     size_t length,
                                     napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] str` : Character buffer representing a UTF16-LE-encoded string.
- `[in] length` : The length of the string in two-byte code units, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result` : A `napi_value` representing a JavaScript String.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript String object from a UTF16-LE-encoded C string

The JavaScript String type is described in Section 6.1.4 of the ECMAScript Language Specification.

napi_create_string_utf8

Added in: v8.0.0

```
napi_status napi_create_string_utf8(napi_env env,
                                    const char* str,
                                    size_t length,
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] str` : Character buffer representing a UTF8-encoded string.
- `[in] length` : The length of the string in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result` : A `napi_value` representing a JavaScript String.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript String object from a UTF8-encoded C string

The JavaScript String type is described in Section 6.1.4 of the ECMAScript Language Specification.

Functions to convert from N-API to C types

napi_get_array_length

Added in: v8.0.0

```
napi_status napi_get_array_length(napi_env env,
                                  napi_value value,
                                  uint32_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing the JavaScript Array whose length is being queried.
- `[out] result` : `uint32` representing length of the array.

Returns `napi_ok` if the API succeeded.

This API returns the length of an array.

Array length is described in Section 22.1.4.1 of the ECMAScript Language Specification.

napi_get_arraybuffer_info

Added in: v8.0.0

```
napi_status napi_get_arraybuffer_info(napi_env env,
                                      napi_value arraybuffer,
                                      void** data,
                                      size_t* byte_length)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] arraybuffer` : `napi_value` representing the ArrayBuffer being queried.
- `[out] data` : The underlying data buffer of the ArrayBuffer.
- `[out] byte_length` : Length in bytes of the underlying data buffer.

Returns `napi_ok` if the API succeeded.

This API is used to retrieve the underlying data buffer of an ArrayBuffer and its length.

WARNING: Use caution while using this API. The lifetime of the underlying data buffer is managed by the ArrayBuffer even after it's returned. A possible safe way to use this API is in conjunction with `napi_create_reference` , which can be used to guarantee control over the lifetime of the ArrayBuffer. It's also safe to use the returned data buffer within the same callback as long as there are no calls to other APIs that might trigger a GC.

napi_get_buffer_info

Added in: v8.0.0

```
napi_status napi_get_buffer_info(napi_env env,
                                 napi_value value,
                                 void** data,
                                 size_t* length)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing the `node::Buffer` being queried.
- `[out] data` : The underlying data buffer of the `node::Buffer` .
- `[out] length` : Length in bytes of the underlying data buffer.

Returns `napi_ok` if the API succeeded.

This API is used to retrieve the underlying data buffer of a `node::Buffer` and it's length.

Warning: Use caution while using this API since the underlying data buffer's lifetime is not guaranteed if it's managed by the VM.

napi_get_prototype

Added in: v8.0.0

```
napi_status napi_get_prototype(napi_env env,
                               napi_value object,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] object` : `napi_value` representing JavaScript Object whose prototype to return. This returns the equivalent of `Object.getPrototypeOf` (which is not the same as the function's `prototype` property).
- `[out] result` : `napi_value` representing prototype of the given object.

Returns `napi_ok` if the API succeeded.

napi_get_typedarray_info

Added in: v8.0.0

```
napi_status napi_get_typedarray_info(napi_env env,
                                     napi_value typedarray,
                                     napi_typedarray_type* type,
                                     size_t* length,
                                     void** data,
                                     napi_value* arraybuffer,
                                     size_t* byte_offset)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] typedarray` : `napi_value` representing the TypedArray whose properties to query.
- `[out] type` : Scalar datatype of the elements within the TypedArray.
- `[out] length` : Number of elements in the TypedArray.
- `[out] data` : The data buffer underlying the typed array.
- `[out] byte_offset` : The byte offset within the data buffer from which to start projecting the TypedArray.

Returns `napi_ok` if the API succeeded.

This API returns various properties of a typed array.

Warning: Use caution while using this API since the underlying data buffer is managed by the VM

napi_get_dataview_info

Added in: v8.3.0

```
napi_status napi_get_dataview_info(napi_env env,
                                   napi_value dataview,
                                   size_t* byte_length,
                                   void** data,
                                   napi_value* arraybuffer,
                                   size_t* byte_offset)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] dataview` : `napi_value` representing the DataView whose properties to query.

- `[out] byte_length` : Number of bytes in the DataView.
- `[out] data` : The data buffer underlying the DataView.
- `[out] arraybuffer` : ArrayBuffer underlying the DataView.
- `[out] byte_offset` : The byte offset within the data buffer from which to start projecting the DataView.

Returns `napi_ok` if the API succeeded.

This API returns various properties of a DataView.

napi_get_value_bool

Added in: v8.0.0

```
napi_status napi_get_value_bool(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript Boolean.
- `[out] result` : C boolean primitive equivalent of the given JavaScript Boolean.

Returns `napi_ok` if the API succeeded. If a non-boolean `napi_value` is passed in it returns `napi_boolean_expected` .

This API returns the C boolean primitive equivalent of the given JavaScript Boolean.

napi_get_value_double

Added in: v8.0.0

```
napi_status napi_get_value_double(napi_env env,
                                  napi_value value,
                                  double* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript Number.
- `[out] result` : C double primitive equivalent of the given JavaScript Number.

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected` .

This API returns the C double primitive equivalent of the given JavaScript Number.

napi_get_value_external

Added in: v8.0.0

```
napi_status napi_get_value_external(napi_env env,
                                    napi_value value,
                                    void** result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript external value.
- `[out] result` : Pointer to the data wrapped by the JavaScript external value.

Returns `napi_ok` if the API succeeded. If a non-external `napi_value` is passed in it returns `napi_invalid_arg` .

This API retrieves the external data pointer that was previously passed to `napi_create_external()` .

napi_get_value_int32

Added in: v8.0.0

```
napi_status napi_get_value_int32(napi_env env,
                                 napi_value value,
                                 int32_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript Number.
- `[out] result` : C int32 primitive equivalent of the given JavaScript Number.

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in `napi_number_expected .

This API returns the C int32 primitive equivalent of the given JavaScript Number. If the number exceeds the range of the 32 bit integer, then the result is truncated to the equivalent of the bottom 32 bits. This can result in a large positive number becoming a negative number if the value is > 2^31 -1.

napi_get_value_int64

Added in: v8.0.0

```
napi_status napi_get_value_int64(napi_env env,
                                 napi_value value,
                                 int64_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript Number.
- `[out] result` : C int64 primitive equivalent of the given JavaScript Number.

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected`.

This API returns the C int64 primitive equivalent of the given JavaScript Number.

napi_get_value_string_latin1

Added in: v8.0.0

```
napi_status napi_get_value_string_latin1(napi_env env,
                                         napi_value value,
                                         char* buf,
                                         size_t bufsize,
                                         size_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript string.
- `[in] buf` : Buffer to write the ISO-8859-1-encoded string into. If NULL is passed in, the length of the string (in bytes) is returned.
- `[in] bufsize` : Size of the destination buffer. When this value is insufficient, the returned string will be truncated.
- `[out] result` : Number of bytes copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non-String `napi_value` is passed in it returns `napi_string_expected`.

This API returns the ISO-8859-1-encoded string corresponding the value passed in.

napi_get_value_string_utf8

Added in: v8.0.0

```
napi_status napi_get_value_string_utf8(napi_env env,
                                       napi_value value,
                                       char* buf,
                                       size_t bufsize,
                                       size_t* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: `napi_value` representing JavaScript string.
- `[in] buf`: Buffer to write the UTF8-encoded string into. If NULL is passed in, the length of the string (in bytes) is returned.
- `[in] bufsize`: Size of the destination buffer. When this value is insufficient, the returned string will be truncated.
- `[out] result`: Number of bytes copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non-String `napi_value` is passed in it returns `napi_string_expected`.

This API returns the UTF8-encoded string corresponding the value passed in.

napi_get_value_string_utf16

Added in: v8.0.0

```
napi_status napi_get_value_string_utf16(napi_env env,
                                        napi_value value,
                                        char16_t* buf,
                                        size_t bufsize,
                                        size_t* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: `napi_value` representing JavaScript string.
- `[in] buf`: Buffer to write the UTF16-LE-encoded string into. If NULL is passed in, the length of the string (in 2-byte code units) is returned.
- `[in] bufsize`: Size of the destination buffer. When this value is insufficient, the returned string will be truncated.
- `[out] result`: Number of 2-byte code units copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non-String `napi_value` is passed in it returns `napi_string_expected`.

This API returns the UTF16-encoded string corresponding the value passed in.

napi_get_value_uint32

Added in: v8.0.0

```
napi_status napi_get_value_uint32(napi_env env,
                                  napi_value value,
                                  uint32_t* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: `napi_value` representing JavaScript Number.
- `[out] result`: C primitive equivalent of the given `napi_value` as a `uint32_t`.

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected`.

This API returns the C primitive equivalent of the given `napi_value` as a `uint32_t`.

Functions to get global instances

napi_get_boolean

Added in: v8.0.0

```
napi_status napi_get_boolean(napi_env env, bool value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The value of the boolean to retrieve.
- `[out] result` : `napi_value` representing JavaScript Boolean singleton to retrieve.

Returns `napi_ok` if the API succeeded.

This API is used to return the JavaScript singleton object that is used to represent the given boolean value

napi_get_global

Added in: v8.0.0

```
napi_status napi_get_global(napi_env env, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing JavaScript Global Object.

Returns `napi_ok` if the API succeeded.

This API returns the global Object.

napi_get_null

Added in: v8.0.0

```
napi_status napi_get_null(napi_env env, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing JavaScript Null Object.

Returns `napi_ok` if the API succeeded.

This API returns the null Object.

napi_get_undefined

Added in: v8.0.0

```
napi_status napi_get_undefined(napi_env env, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing JavaScript Undefined value.

Returns `napi_ok` if the API succeeded.

This API returns the Undefined object.

Working with JavaScript Values - Abstract Operations

N-API exposes a set of APIs to perform some abstract operations on JavaScript values. Some of these operations are documented under Section 7 of the ECMAScript Language Specification .

These APIs support doing one of the following:

1. Coerce JavaScript values to specific JavaScript types (such as Number or String)
2. Check the type of a JavaScript value
3. Check for equality between two JavaScript values

napi_coerce_to_bool

Added in: v8.0.0

```
napi_status napi_coerce_to_bool(napi_env env,
                                napi_value value,
                                napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to coerce.
- `[out] result` : `napi_value` representing the coerced JavaScript Boolean.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation ToBoolean as defined in Section 7.1.2 of the ECMAScript Language Specification. This API can be re-entrant if getters are defined on the passed-in Object.

napi_coerce_to_number

Added in: v8.0.0

```
napi_status napi_coerce_to_number(napi_env env,
                                  napi_value value,
                                  napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to coerce.
- `[out] result` : `napi_value` representing the coerced JavaScript Number.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation ToNumber as defined in Section 7.1.3 of the ECMAScript Language Specification. This API can be re-entrant if getters are defined on the passed-in Object.

napi_coerce_to_object

Added in: v8.0.0

```
napi_status napi_coerce_to_object(napi_env env,
                                  napi_value value,
                                  napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to coerce.
- `[out] result` : `napi_value` representing the coerced JavaScript Object.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation ToObject as defined in Section 7.1.13 of the ECMAScript Language Specification. This API can be re-entrant if getters are defined on the passed-in Object.

napi_coerce_to_string

Added in: v8.0.0

```
napi_status napi_coerce_to_string(napi_env env,
                                  napi_value value,
                                  napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to coerce.
- `[out] result` : `napi_value` representing the coerced JavaScript String.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation ToString as defined in Section 7.1.13 of the ECMAScript Language Specification. This API can be re-entrant if getters are defined on the passed-in Object.

napi_typeof

Added in: v8.0.0

```
napi_status napi_typeof(napi_env env, napi_value value, napi_valuetype* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value whose type to query.
- `[out] result` : The type of the JavaScript value.

Returns `napi_ok` if the API succeeded.

- `napi_invalid_arg` if the type of `value` is not a known ECMAScript type and `value` is not an External value.

This API represents behavior similar to invoking the `typeof` Operator on the object as defined in Section 12.5.5 of the ECMAScript Language Specification. However, it has support for detecting an External value. If `value` has a type that is invalid, an error is returned.

napi_instanceof

Added in: v8.0.0

```
napi_status napi_instanceof(napi_env env,
                            napi_value object,
                            napi_value constructor,
                            bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] object` : The JavaScript value to check.
- `[in] constructor` : The JavaScript function object of the constructor function to check against.
- `[out] result` : Boolean that is set to true if `object instanceof constructor` is true.

Returns `napi_ok` if the API succeeded.

This API represents invoking the `instanceof` Operator on the object as defined in Section 12.10.4 of the ECMAScript Language Specification.

napi_is_array

Added in: v8.0.0

```
napi_status napi_is_array(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given object is an array.

Returns `napi_ok` if the API succeeded.

This API represents invoking the `IsArray` operation on the object as defined in Section 7.2.2 of the ECMAScript Language Specification.

napi_is_arraybuffer

Added in: v8.0.0

```
napi_status napi_is_arraybuffer(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given object is an ArrayBuffer.

Returns `napi_ok` if the API succeeded.

This API checks if the Object passed in is an array buffer.

napi_is_buffer

Added in: v8.0.0

```
napi_status napi_is_buffer(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a `node::Buffer` object.

Returns `napi_ok` if the API succeeded.

This API checks if the Object passed in is a buffer.

napi_is_error

Added in: v8.0.0

```
napi_status napi_is_error(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents an Error object.

Returns `napi_ok` if the API succeeded.

This API checks if the Object passed in is an Error.

napi_is_typedarray

Added in: v8.0.0

```
napi_status napi_is_typedarray(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a TypedArray.

Returns `napi_ok` if the API succeeded.

This API checks if the Object passed in is a typed array.

napi_is_dataview

Added in: v8.3.0

```
napi_status napi_is_dataview(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a DataView.

Returns `napi_ok` if the API succeeded.

This API checks if the Object passed in is a DataView.

napi_strict_equals

Added in: v8.0.0

```
napi_status napi_strict_equals(napi_env env,
                               napi_value lhs,
                               napi_value rhs,
                               bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] lhs` : The JavaScript value to check.
- `[in] rhs` : The JavaScript value to check against.
- `[out] result` : Whether the two `napi_value` objects are equal.

Returns `napi_ok` if the API succeeded.

This API represents the invocation of the Strict Equality algorithm as defined in Section 7.2.14 of the ECMAScript Language Specification.

Working with JavaScript Properties

N-API exposes a set of APIs to get and set properties on JavaScript objects. Some of these types are documented under Section 7 of the ECMAScript Language Specification .

Properties in JavaScript are represented as a tuple of a key and a value. Fundamentally, all property keys in N-API can be represented in one of the following forms:

- Named: a simple UTF8-encoded string
- Integer-Indexed: an index value represented by `uint32_t`
- JavaScript value: these are represented in N-API by `napi_value` . This can be a `napi_value` representing a String, Number, or Symbol.

N-API values are represented by the type `napi_value` . Any N-API call that requires a JavaScript value takes in a `napi_value` . However, it's the caller's responsibility to make sure that the `napi_value` in question is of the JavaScript type expected by the API.

The APIs documented in this section provide a simple interface to get and set properties on arbitrary JavaScript objects represented by `napi_value` .

For instance, consider the following JavaScript code snippet:

```
const obj = {};
obj.myProp = 123;
```

The equivalent can be done using N-API values with the following snippet:

```
napi_status status = napi_generic_failure;

// const obj = {}
napi_value obj, value;
status = napi_create_object(env, &obj);
if (status != napi_ok) return status;

// Create a napi_value for 123
status = napi_create_int32(env, 123, &value);
if (status != napi_ok) return status;

// obj.myProp = 123
status = napi_set_named_property(env, obj, "myProp", value);
if (status != napi_ok) return status;
```

Indexed properties can be set in a similar manner. Consider the following JavaScript snippet:

```
const arr = [];
arr[123] = 'hello';
```

The equivalent can be done using N-API values with the following snippet:

```
napi_status status = napi_generic_failure;

// const arr = [];
napi_value arr, value;
status = napi_create_array(env, &arr);
if (status != napi_ok) return status;

// Create a napi_value for 'hello'
status = napi_create_string_utf8(env, "hello", NAPI_AUTO_LENGTH, &value);
if (status != napi_ok) return status;

// arr[123] = 'hello';
status = napi_set_element(env, arr, 123, value);
if (status != napi_ok) return status;
```

Properties can be retrieved using the APIs described in this section. Consider the following JavaScript snippet:

```
const arr = [];
const value = arr[123];
```

The following is the approximate equivalent of the N-API counterpart:

```
napi_status status = napi_generic_failure;

// const arr = []
napi_value arr, value;
status = napi_create_array(env, &arr);
if (status != napi_ok) return status;

// const value = arr[123]
status = napi_get_element(env, arr, 123, &value);
if (status != napi_ok) return status;
```

Finally, multiple properties can also be defined on an object for performance reasons. Consider the following JavaScript:

```
const obj = {};
Object.defineProperties(obj, {
  'foo': { value: 123, writable: true, configurable: true, enumerable: true },
  'bar': { value: 456, writable: true, configurable: true, enumerable: true }
});
```

The following is the approximate equivalent of the N-API counterpart:

```
napi_status status = napi_status_generic_failure;

// const obj = {};
napi_value obj;
status = napi_create_object(env, &obj);
if (status != napi_ok) return status;

// Create napi_values for 123 and 456
napi_value fooValue, barValue;
status = napi_create_int32(env, 123, &fooValue);
if (status != napi_ok) return status;
status = napi_create_int32(env, 456, &barValue);
if (status != napi_ok) return status;

// Set the properties
napi_property_descriptor descriptors[] = {
  { "foo", NULL, 0, 0, 0, fooValue, napi_default, 0 },
  { "bar", NULL, 0, 0, 0, barValue, napi_default, 0 }
}
status = napi_define_properties(env,
                                obj,
                                sizeof(descriptors) / sizeof(descriptors[0]),
                                descriptors);
if (status != napi_ok) return status;
```

Structures

napi_property_attributes

```
typedef enum {
  napi_default = 0,
  napi_writable = 1 << 0,
  napi_enumerable = 1 << 1,
  napi_configurable = 1 << 2,

  // Used with napi_define_class to distinguish static properties
  // from instance properties. Ignored by napi_define_properties.
  napi_static = 1 << 10,
} napi_property_attributes;
```

`napi_property_attributes` are flags used to control the behavior of properties set on a JavaScript object. Other than `napi_static` they correspond to the attributes listed in Section 6.1.7.1 of the ECMAScript Language Specification . They can be one or more of the following bitflags:

- `napi_default` - Used to indicate that no explicit attributes are set on the given property. By default, a property is read only, not enumerable and not configurable.

- `napi_writable` - Used to indicate that a given property is writable.
- `napi_enumerable` - Used to indicate that a given property is enumerable.
- `napi_configurable` - Used to indicate that a given property is configurable, as defined in Section 6.1.7.1 of the ECMAScript Language Specification.
- `napi_static` - Used to indicate that the property will be defined as a static property on a class as opposed to an instance property, which is the default. This is used only by `napi_define_class`. It is ignored by `napi_define_properties`.

napi_property_descriptor

```
typedef struct {
  // One of utf8name or name should be NULL.
  const char* utf8name;
  napi_value name;

  napi_callback method;
  napi_callback getter;
  napi_callback setter;
  napi_value value;

  napi_property_attributes attributes;
  void* data;
} napi_property_descriptor;
```

- `utf8name`: Optional String describing the key for the property, encoded as UTF8. One of `utf8name` or `name` must be provided for the property.
- `name`: Optional napi_value that points to a JavaScript string or symbol to be used as the key for the property. One of `utf8name` or `name` must be provided for the property.
- `value`: The value that's retrieved by a get access of the property if the property is a data property. If this is passed in, set `getter`, `setter`, `method` and `data` to `NULL` (since these members won't be used).
- `getter`: A function to call when a get access of the property is performed. If this is passed in, set `value` and `method` to `NULL` (since these members won't be used). The given function is called implicitly by the runtime when the property is accessed from JavaScript code (or if a get on the property is performed using a N-API call).
- `setter`: A function to call when a set access of the property is performed. If this is passed in, set `value` and `method` to `NULL` (since these members won't be used). The given function is called implicitly by the runtime when the property is set from JavaScript code (or if a set on the property is performed using a N-API call).
- `method`: Set this to make the property descriptor object's `value` property to be a JavaScript function represented by `method`. If this is passed in, set `value`, `getter` and `setter` to `NULL` (since these members won't be used).
- `data`: The callback data passed into `method`, `getter` and `setter` if this function is invoked.
- `attributes`: The attributes associated with the particular property. See `napi_property_attributes`.

Functions

napi_get_property_names

Added in: v8.0.0

```
napi_status napi_get_property_names(napi_env env,
                                    napi_value object,
                                    napi_value* result);
```

- `[in] env`: The environment that the N-API call is invoked under.
- `[in] object`: The object from which to retrieve the properties.
- `[out] result`: A `napi_value` representing an array of JavaScript values that represent the property names of the object. The API can be used to iterate over `result` using `napi_get_array_length` and `napi_get_element`.

Returns `napi_ok` if the API succeeded.

This API returns the array of properties for the Object passed in

napi_set_property

Added in: v8.0.0

```
napi_status napi_set_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              napi_value value);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object on which to set the property.
- `[in] key` : The name of the property to set.
- `[in] value` : The property value.

Returns `napi_ok` if the API succeeded.

This API set a property on the Object passed in.

napi_get_property

Added in: v8.0.0

```
napi_status napi_get_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              napi_value* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object from which to retrieve the property.
- `[in] key` : The name of the property to retrieve.
- `[out] result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This API gets the requested property from the Object passed in.

napi_has_property

Added in: v8.0.0

```
napi_status napi_has_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              bool* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object to query.
- `[in] key` : The name of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API checks if the Object passed in has the named property.

napi_delete_property

Added in: v8.2.0

```
napi_status napi_delete_property(napi_env env,
                                 napi_value object,
                                 napi_value key,
                                 bool* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object to query.
- `[in] key` : The name of the property to delete.
- `[out] result` : Whether the property deletion succeeded or not. `result` can optionally be ignored by passing `NULL` .

Returns `napi_ok` if the API succeeded.

This API attempts to delete the `key` own property from `object` .

napi_has_own_property

Added in: v8.2.0

```
napi_status napi_has_own_property(napi_env env,
                                  napi_value object,
                                  napi_value key,
                                  bool* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object to query.
- `[in] key` : The name of the own property whose existence to check.
- `[out] result` : Whether the own property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API checks if the Object passed in has the named own property. `key` must be a string or a Symbol, or an error will be thrown. N-API will not perform any conversion between data types.

napi_set_named_property

Added in: v8.0.0

```
napi_status napi_set_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    napi_value value);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object on which to set the property.
- `[in] utf8Name` : The name of the property to set.
- `[in] value` : The property value.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling napi_set_property with a `napi_value` created from the string passed in as `utf8Name`

napi_get_named_property

Added in: v8.0.0

```
napi_status napi_get_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    napi_value* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object from which to retrieve the property.
- `[in] utf8Name` : The name of the property to get.
- `[out] result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling napi_get_property with a `napi_value` created from the string passed in as `utf8Name`

napi_has_named_property

Added in: v8.0.0

```
napi_status napi_has_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    bool* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object to query.
- `[in] utf8Name` : The name of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling napi_has_property with a `napi_value` created from the string passed in as `utf8Name`

napi_set_element

Added in: v8.0.0

```
napi_status napi_set_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             napi_value value);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object from which to set the properties.
- `[in] index` : The index of the property to set.
- `[in] value` : The property value.

Returns `napi_ok` if the API succeeded.

This API sets and element on the Object passed in.

napi_get_element

Added in: v8.0.0

```
napi_status napi_get_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             napi_value* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object from which to retrieve the property.
- `[in] index` : The index of the property to get.
- `[out] result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This API gets the element at the requested index.

napi_has_element

Added in: v8.0.0

```
napi_status napi_has_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             bool* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object to query.
- `[in] index` : The index of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API returns if the Object passed in has an element at the requested index.

napi_delete_element

Added in: v8.2.0

```
napi_status napi_delete_element(napi_env env,
                                napi_value object,
                                uint32_t index,
                                bool* result);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object to query.
- `[in] index` : The index of the property to delete.
- `[out] result` : Whether the element deletion succeeded or not. `result` can optionally be ignored by passing `NULL` .

Returns `napi_ok` if the API succeeded.

This API attempts to delete the specified `index` from `object` .

napi_define_properties

Added in: v8.0.0

```
napi_status napi_define_properties(napi_env env,
                                   napi_value object,
                                   size_t property_count,
                                   const napi_property_descriptor* properties);
```

- `[in] env` : The environment that the N-API call is invoked under.
- `[in] object` : The object from which to retrieve the properties.
- `[in] property_count` : The number of elements in the `properties` array.
- `[in] properties` : The array of property descriptors.

Returns `napi_ok` if the API succeeded.

This method allows the efficient definition of multiple properties on a given object. The properties are defined using property descriptors (See `napi_property_descriptor`). Given an array of such property descriptors, this API will set the properties on the object one at a time, as defined by DefineOwnProperty (described in Section 9.1.6 of the ECMA262 specification).

Working with JavaScript Functions

N-API provides a set of APIs that allow JavaScript code to call back into native code. N-API APIs that support calling back into native code take in a callback functions represented by the `napi_callback` type. When the JavaScript VM calls back to native code, the `napi_callback` function provided is invoked. The APIs documented in this section allow the callback function to do the following:

- Get information about the context in which the callback was invoked.
- Get the arguments passed into the callback.
- Return a `napi_value` back from the callback.

Additionally, N-API provides a set of functions which allow calling JavaScript functions from native code. One can either call a function like a regular JavaScript function call, or as a constructor function.

napi_call_function

Added in: v8.0.0

```
napi_status napi_call_function(napi_env env,
                               napi_value recv,
                               napi_value func,
                               int argc,
                               const napi_value* argv,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] recv` : The `this` object passed to the called function.
- `[in] func` : `napi_value` representing the JavaScript function to be invoked.
- `[in] argc` : The count of elements in the `argv` array.
- `[in] argv` : Array of `napi_values` representing JavaScript values passed in as arguments to the function.
- `[out] result` : `napi_value` representing the JavaScript object returned.

Returns `napi_ok` if the API succeeded.

This method allows a JavaScript function object to be called from a native add-on. This is the primary mechanism of calling back *from* the add-on's native code *into* JavaScript. For the special case of calling into JavaScript after an async operation, see `napi_make_callback` .

A sample use case might look as follows. Consider the following JavaScript snippet:

```
function AddTwo(num) {
  return num + 2;
}
```

Then, the above function can be invoked from a native add-on using the following code:

```
// Get the function named "AddTwo" on the global object
napi_value global, add_two, arg;
napi_status status = napi_get_global(env, &global);
if (status != napi_ok) return;

status = napi_get_named_property(env, global, "AddTwo", &add_two);
if (status != napi_ok) return;

// const arg = 1337
status = napi_create_int32(env, 1337, &arg);
if (status != napi_ok) return;

napi_value* argv = &arg;
size_t argc = 1;

// AddTwo(arg);
napi_value return_val;
status = napi_call_function(env, global, add_two, argc, argv, &return_val);
if (status != napi_ok) return;

// Convert the result back to a native type
int32_t result;
status = napi_get_value_int32(env, return_val, &result);
if (status != napi_ok) return;
```

napi_create_function

Added in: v8.0.0

```
napi_status napi_create_function(napi_env env,
                                 const char* utf8name,
                                 napi_callback cb,
                                 void* data,
                                 napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] utf8Name` : The name of the function encoded as UTF8. This is visible within JavaScript as the new function object's `name` property.
- `[in] cb` : The native function which should be called when this function object is invoked.
- `[in] data` : User-provided data context. This will be passed back into the function when invoked later.
- `[out] result` : `napi_value` representing the JavaScript function object for the newly created function.

Returns `napi_ok` if the API succeeded.

This API allows an add-on author to create a function object in native code. This is the primary mechanism to allow calling *into* the add-on's native code *from* JavaScript.

The newly created function is not automatically visible from script after this call. Instead, a property must be explicitly set on any object that is visible to JavaScript, in order for the function to be accessible from script.

In order to expose a function as part of the add-on's module exports, set the newly created function on the exports object. A sample module

might look as follows:

```
napi_value SayHello(napi_env env, napi_callback_info info) {
  printf("Hello\n");
  return NULL;
}

napi_value Init(napi_env env, napi_value exports) {
  napi_status status;

  napi_value fn;
  status = napi_create_function(env, NULL, 0, SayHello, NULL, &fn);
  if (status != napi_ok) return NULL;

  status = napi_set_named_property(env, exports, "sayHello", fn);
  if (status != napi_ok) return NULL;

  return exports;
}

NAPI_MODULE(NODE_GYP_MODULE_NAME, Init)
```

Given the above code, the add-on can be used from JavaScript as follows:

```
const myaddon = require('./addon');
myaddon.sayHello();
```

The string passed to require is not necessarily the name passed into `NAPI_MODULE` in the earlier snippet but the name of the target in `binding.gyp` responsible for creating the `.node` file.

napi_get_cb_info

Added in: v8.0.0

```
napi_status napi_get_cb_info(napi_env env,
                             napi_callback_info cbinfo,
                             size_t* argc,
                             napi_value* argv,
                             napi_value* thisArg,
                             void** data)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] cbinfo` : The callback info passed into the callback function.
- `[in-out] argc` : Specifies the size of the provided `argv` array and receives the actual count of arguments.
- `[out] argv` : Buffer to which the `napi_value` representing the arguments are copied. If there are more arguments than the provided count, only the requested number of arguments are copied. If there are fewer arguments provided than claimed, the rest of `argv` is filled with `napi_value` values that represent `undefined`.
- `[out] this` : Receives the JavaScript `this` argument for the call.
- `[out] data` : Receives the data pointer for the callback.

Returns `napi_ok` if the API succeeded.

This method is used within a callback function to retrieve details about the call like the arguments and the `this` pointer from a given callback info.

napi_get_new_target

Added in: v8.6.0

```
napi_status napi_get_new_target(napi_env env,
                                napi_callback_info cbinfo,
                                napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] cbinfo` : The callback info passed into the callback function.
- `[out] result` : The `new.target` of the constructor call.

Returns `napi_ok` if the API succeeded.

This API returns the `new.target` of the constructor call. If the current callback is not a constructor call, the result is `NULL`.

napi_new_instance

Added in: v8.0.0

```
napi_status napi_new_instance(napi_env env,
                              napi_value cons,
                              size_t argc,
                              napi_value* argv,
                              napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] cons` : `napi_value` representing the JavaScript function to be invoked as a constructor.
- `[in] argc` : The count of elements in the `argv` array.
- `[in] argv` : Array of JavaScript values as `napi_value` representing the arguments to the constructor.
- `[out] result` : `napi_value` representing the JavaScript object returned, which in this case is the constructed object.

This method is used to instantiate a new JavaScript value using a given `napi_value` that represents the constructor for the object. For example, consider the following snippet:

```
function MyObject(param) {
  this.param = param;
}

const arg = 'hello';
const value = new MyObject(arg);
```

The following can be approximated in N-API using the following snippet:

```
// Get the constructor function MyObject
napi_value global, constructor, arg, value;
napi_status status = napi_get_global(env, &global);
if (status != napi_ok) return;

status = napi_get_named_property(env, global, "MyObject", &constructor);
if (status != napi_ok) return;

// const arg = "hello"
status = napi_create_string_utf8(env, "hello", NAPI_AUTO_LENGTH, &arg);
if (status != napi_ok) return;

napi_value* argv = &arg;
size_t argc = 1;

// const value = new MyObject(arg)
status = napi_new_instance(env, constructor, argc, argv, &value);
```

Returns `napi_ok` if the API succeeded.

Object Wrap

N-API offers a way to "wrap" C++ classes and instances so that the class constructor and methods can be called from JavaScript.

1. The napi_define_class API defines a JavaScript class with constructor, static properties and methods, and instance properties and methods that correspond to the C++ class.
2. When JavaScript code invokes the constructor, the constructor callback uses napi_wrap to wrap a new C++ instance in a JavaScript object, then returns the wrapper object.
3. When JavaScript code invokes a method or property accessor on the class, the corresponding `napi_callback` C++ function is invoked. For an instance callback, napi_unwrap obtains the C++ instance that is the target of the call.

For wrapped objects it may be difficult to distinguish between a function called on a class prototype and a function called on an instance of a class. A common pattern used to address this problem is to save a persistent reference to the class constructor for later `instanceof` checks.

As an example:

```
napi_value MyClass_constructor = NULL;
status = napi_get_reference_value(env, MyClass::es_constructor, &MyClass_constructor);
assert(napi_ok == status);
bool is_instance = false;
status = napi_instanceof(env, es_this, MyClass_constructor, &is_instance);
assert(napi_ok == status);
if (is_instance) {
  // napi_unwrap() ...
} else {
  // otherwise...
}
```

The reference must be freed once it is no longer needed.

napi_define_class

Added in: v8.0.0

```
napi_status napi_define_class(napi_env env,
                              const char* utf8name,
                              size_t length,
                              napi_callback constructor,
                              void* data,
                              size_t property_count,
                              const napi_property_descriptor* properties,
                              napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] utf8name` : Name of the JavaScript constructor function; this is not required to be the same as the C++ class name, though it is recommended for clarity.
- `[in] length` : The length of the utf8name in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] constructor` : Callback function that handles constructing instances of the class. (This should be a static method on the class, not an actual C++ constructor function.)
- `[in] data` : Optional data to be passed to the constructor callback as the `data` property of the callback info.
- `[in] property_count` : Number of items in the `properties` array argument.
- `[in] properties` : Array of property descriptors describing static and instance data properties, accessors, and methods on the class See `napi_property_descriptor` .
- `[out] result` : A `napi_value` representing the constructor function for the class.

Returns `napi_ok` if the API succeeded.

Defines a JavaScript class that corresponds to a C++ class, including:

- A JavaScript constructor function that has the class name and invokes the provided C++ constructor callback.
- Properties on the constructor function corresponding to *static* data properties, accessors, and methods of the C++ class (defined by property descriptors with the `napi_static` attribute).
- Properties on the constructor function's `prototype` object corresponding to *non-static* data properties, accessors, and methods of the C++ class (defined by property descriptors without the `napi_static` attribute).

The C++ constructor callback should be a static method on the class that calls the actual class constructor, then wraps the new C++ instance in a JavaScript object, and returns the wrapper object. See `napi_wrap()` for details.

The JavaScript constructor function returned from `napi_define_class` is often saved and used later, to construct new instances of the class from native code, and/or check whether provided values are instances of the class. In that case, to prevent the function value from being garbage-collected, create a persistent reference to it using `napi_create_reference` and ensure the reference count is kept >= 1.

napi_wrap

Added in: v8.0.0

```
napi_status napi_wrap(napi_env env,
                      napi_value js_object,
                      void* native_object,
                      napi_finalize finalize_cb,
                      void* finalize_hint,
                      napi_ref* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The JavaScript object that will be the wrapper for the native object. This object *must* have been created from the `prototype` of a constructor that was created using `napi_define_class()` .
- `[in] native_object` : The native instance that will be wrapped in the JavaScript object.
- `[in] finalize_cb` : Optional native callback that can be used to free the native instance when the JavaScript object is ready for garbage-collection.
- `[in] finalize_hint` : Optional contextual hint that is passed to the finalize callback.

- `[out] result`: Optional reference to the wrapped object.

Returns `napi_ok` if the API succeeded.

Wraps a native instance in a JavaScript object. The native instance can be retrieved later using `napi_unwrap()`.

When JavaScript code invokes a constructor for a class that was defined using `napi_define_class()`, the `napi_callback` for the constructor is invoked. After constructing an instance of the native class, the callback must then call `napi_wrap()` to wrap the newly constructed instance in the already-created JavaScript object that is the `this` argument to the constructor callback. (That `this` object was created from the constructor function's `prototype`, so it already has definitions of all the instance properties and methods.)

Typically when wrapping a class instance, a finalize callback should be provided that simply deletes the native instance that is received as the `data` argument to the finalize callback.

The optional returned reference is initially a weak reference, meaning it has a reference count of 0. Typically this reference count would be incremented temporarily during async operations that require the instance to remain valid.

Caution: The optional returned reference (if obtained) should be deleted via napi_delete_reference ONLY in response to the finalize callback invocation. (If it is deleted before then, then the finalize callback may never be invoked.) Therefore, when obtaining a reference a finalize callback is also required in order to enable correct proper of the reference.

This API may modify the prototype chain of the wrapper object. Afterward, additional manipulation of the wrapper's prototype chain may cause `napi_unwrap()` to fail.

Calling napi_wrap() a second time on an object will return an error. To associate another native instance with the object, use napi_remove_wrap() first.

napi_unwrap

Added in: v8.0.0

```
napi_status napi_unwrap(napi_env env,
                        napi_value js_object,
                        void** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] js_object`: The object associated with the native instance.
- `[out] result`: Pointer to the wrapped native instance.

Returns `napi_ok` if the API succeeded.

Retrieves a native instance that was previously wrapped in a JavaScript object using `napi_wrap()`.

When JavaScript code invokes a method or property accessor on the class, the corresponding `napi_callback` is invoked. If the callback is for an instance method or accessor, then the `this` argument to the callback is the wrapper object; the wrapped C++ instance that is the target of the call can be obtained then by calling `napi_unwrap()` on the wrapper object.

napi_remove_wrap

Added in: v8.5.0

```
napi_status napi_remove_wrap(napi_env env,
                             napi_value js_object,
                             void** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] js_object`: The object associated with the native instance.
- `[out] result`: Pointer to the wrapped native instance.

Returns `napi_ok` if the API succeeded.

Retrieves a native instance that was previously wrapped in the JavaScript object `js_object` using `napi_wrap()` and removes the wrapping, thereby restoring the JavaScript object's prototype chain. If a finalize callback was associated with the wrapping, it will no longer be called when the JavaScript object becomes garbage-collected.

Simple Asynchronous Operations

Addon modules often need to leverage async helpers from libuv as part of their implementation. This allows them to schedule work to be executed asynchronously so that their methods can return in advance of the work being completed. This is important in order to allow them to avoid blocking overall execution of the Node.js application.

N-API provides an ABI-stable interface for these supporting functions which covers the most common asynchronous use cases.

N-API defines the `napi_work` structure which is used to manage asynchronous workers. Instances are created/deleted with napi_create_async_work and napi_delete_async_work.

The `execute` and `complete` callbacks are functions that will be invoked when the executor is ready to execute and when it completes its task respectively. These functions implement the following interfaces:

```
typedef void (*napi_async_execute_callback)(napi_env env,
                                            void* data);
typedef void (*napi_async_complete_callback)(napi_env env,
                                             napi_status status,
                                             void* data);
```

When these methods are invoked, the `data` parameter passed will be the addon-provided void* data that was passed into the `napi_create_async_work` call.

Once created the async worker can be queued for execution using the napi_queue_async_work function:

```
napi_status napi_queue_async_work(napi_env env,
                                  napi_async_work work);
```

napi_cancel_async_work can be used if the work needs to be cancelled before the work has started execution.

After calling napi_cancel_async_work, the `complete` callback will be invoked with a status value of `napi_cancelled`. The work should not be deleted before the `complete` callback invocation, even when it was cancelled.

napi_create_async_work

▸ History

```
napi_status napi_create_async_work(napi_env env,
                                   napi_value async_resource,
                                   napi_value async_resource_name,
                                   napi_async_execute_callback execute,
                                   napi_async_complete_callback complete,
                                   void* data,
                                   napi_async_work* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] async_resource`: An optional object associated with the async work that will be passed to possible async_hooks init hooks.
- `[in] async_resource_name`: Identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- `[in] execute`: The native function which should be called to execute the logic asynchronously. The given function is called from a worker pool thread and can execute in parallel with the main event loop thread.
- `[in] complete`: The native function which will be called when the asynchronous logic is completed or is cancelled. The given function is

called from the main event loop thread.

- `[in] data` : User-provided data context. This will be passed back into the execute and complete functions.
- `[out] result` : `napi_async_work*` which is the handle to the newly created async work.

Returns `napi_ok` if the API succeeded.

This API allocates a work object that is used to execute logic asynchronously. It should be freed using `napi_delete_async_work` once the work is no longer required.

`async_resource_name` should be a null-terminated, UTF-8-encoded string.

The `async_resource_name` identifier is provided by the user and should be representative of the type of async work being performed. It is also recommended to apply namespacing to the identifier, e.g. by including the module name. See the `async_hooks` documentation for more information.

napi_delete_async_work

Added in: v8.0.0

```
napi_status napi_delete_async_work(napi_env env,
                                   napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work` .

Returns `napi_ok` if the API succeeded.

This API frees a previously allocated work object.

This API can be called even if there is a pending JavaScript exception.

napi_queue_async_work

Added in: v8.0.0

```
napi_status napi_queue_async_work(napi_env env,
                                  napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work` .

Returns `napi_ok` if the API succeeded.

This API requests that the previously allocated work be scheduled for execution.

napi_cancel_async_work

Added in: v8.0.0

```
napi_status napi_cancel_async_work(napi_env env,
                                   napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work` .

Returns `napi_ok` if the API succeeded.

This API cancels queued work if it has not yet been started. If it has already started executing, it cannot be cancelled and `napi_generic_failure` will be returned. If successful, the `complete` callback will be invoked with a status value of `napi_cancelled` . The work should not be deleted before the `complete` callback invocation, even if it has been successfully cancelled.

This API can be called even if there is a pending JavaScript exception.

Custom Asynchronous Operations

The simple asynchronous work APIs above may not be appropriate for every scenario. When using any other asynchronous mechanism, the following APIs are necessary to ensure an asynchronous operation is properly tracked by the runtime.

napi_async_init

Added in: v8.6.0

```
napi_status napi_async_init(napi_env env,
                            napi_value async_resource,
                            napi_value async_resource_name,
                            napi_async_context* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] async_resource` : An optional object associated with the async work that will be passed to possible `async_hooks` init hooks .
- `[in] async_resource_name` : Identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- `[out] result` : The initialized async context.

Returns `napi_ok` if the API succeeded.

napi_async_destroy

Added in: v8.6.0

```
napi_status napi_async_destroy(napi_env env,
                               napi_async_context async_context);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] async_context` : The async context to be destroyed.

Returns `napi_ok` if the API succeeded.

This API can be called even if there is a pending JavaScript exception.

napi_make_callback

▸ History

```
napi_status napi_make_callback(napi_env env,
                               napi_async_context async_context,
                               napi_value recv,
                               napi_value func,
                               int argc,
                               const napi_value* argv,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] async_context` : Context for the async operation that is invoking the callback. This should normally be a value previously obtained from napi_async_init . However `NULL` is also allowed, which indicates the current async context (if any) is to be used for the callback.
- `[in] recv` : The `this` object passed to the called function.
- `[in] func` : `napi_value` representing the JavaScript function to be invoked.
- `[in] argc` : The count of elements in the `argv` array.

- [in] argv : Array of JavaScript values as napi_value representing the arguments to the function.
- [out] result : napi_value representing the JavaScript object returned.

Returns napi_ok if the API succeeded.

This method allows a JavaScript function object to be called from a native add-on. This API is similar to napi_call_function. However, it is used to call *from* native code back *into* JavaScript *after* returning from an async operation (when there is no other script on the stack). It is a fairly simple wrapper around node::MakeCallback.

Note it is *not* necessary to use napi_make_callback from within a napi_async_complete_callback; in that situation the callback's async context has already been set up, so a direct call to napi_call_function is sufficient and appropriate. Use of the napi_make_callback function may be required when implementing custom async behavior that does not use napi_create_async_work.

napi_open_callback_scope

Added in: v9.6.0

```
NAPI_EXTERN napi_status napi_open_callback_scope(napi_env env,
                                                 napi_value resource_object,
                                                 napi_async_context context,
                                                 napi_callback_scope* result)
```

- [in] env : The environment that the API is invoked under.
- [in] resource_object : An optional object associated with the async work that will be passed to possible async_hooks init hooks.
- [in] context : Context for the async operation that is invoking the callback. This should be a value previously obtained from napi_async_init.
- [out] result : The newly created scope.

There are cases (for example resolving promises) where it is necessary to have the equivalent of the scope associated with a callback in place when making certain N-API calls. If there is no other script on the stack the napi_open_callback_scope and napi_close_callback_scope functions can be used to open/close the required scope.

napi_close_callback_scope

Added in: v9.6.0

```
NAPI_EXTERN napi_status napi_close_callback_scope(napi_env env,
                                                  napi_callback_scope scope)
```

- [in] env : The environment that the API is invoked under.
- [in] scope : The scope to be closed.

This API can be called even if there is a pending JavaScript exception.

Version Management

napi_get_node_version

Added in: v8.4.0

```
typedef struct {
  uint32_t major;
  uint32_t minor;
  uint32_t patch;
  const char* release;
} napi_node_version;

napi_status napi_get_node_version(napi_env env,
                                  const napi_node_version** version);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] version` : A pointer to version information for Node itself.

Returns `napi_ok` if the API succeeded.

This function fills the `version` struct with the major, minor, and patch version of Node.js that is currently running, and the `release` field with the value of `process.release.name` .

The returned buffer is statically allocated and does not need to be freed.

napi_get_version

Added in: v8.0.0

```
napi_status napi_get_version(napi_env env,
                             uint32_t* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : The highest version of N-API supported.

Returns `napi_ok` if the API succeeded.

This API returns the highest N-API version supported by the Node.js runtime. N-API is planned to be additive such that newer releases of Node.js may support additional API functions. In order to allow an addon to use a newer function when running with versions of Node.js that support it, while providing fallback behavior when running with Node.js versions that don't support it:

- Call `napi_get_version()` to determine if the API is available.
- If available, dynamically load a pointer to the function using `uv_dlsym()` .
- Use the dynamically loaded pointer to invoke the function.
- If the function is not available, provide an alternate implementation that does not use the function.

Memory Management

napi_adjust_external_memory

Added in: v8.5.0

```
NAPI_EXTERN napi_status napi_adjust_external_memory(napi_env env,
                                                    int64_t change_in_bytes,
                                                    int64_t* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] change_in_bytes` : The change in externally allocated memory that is kept alive by JavaScript objects.
- `[out] result` : The adjusted value

Returns `napi_ok` if the API succeeded.

This function gives V8 an indication of the amount of externally allocated memory that is kept alive by JavaScript objects (i.e. a JavaScript

object that points to its own memory allocated by a native module). Registering externally allocated memory will trigger global garbage collections more often than it would otherwise.

Promises

N-API provides facilities for creating `Promise` objects as described in Section 25.4 of the ECMA specification. It implements promises as a pair of objects. When a promise is created by `napi_create_promise()`, a "deferred" object is created and returned alongside the `Promise`. The deferred object is bound to the created `Promise` and is the only means to resolve or reject the `Promise` using `napi_resolve_deferred()` or `napi_reject_deferred()`. The deferred object that is created by `napi_create_promise()` is freed by `napi_resolve_deferred()` or `napi_reject_deferred()`. The `Promise` object may be returned to JavaScript where it can be used in the usual fashion.

For example, to create a promise and pass it to an asynchronous worker:

```
napi_deferred deferred;
napi_value promise;
napi_status status;

// Create the promise.
status = napi_create_promise(env, &deferred, &promise);
if (status != napi_ok) return NULL;

// Pass the deferred to a function that performs an asynchronous action.
do_something_asynchronous(deferred);

// Return the promise to JS
return promise;
```

The above function `do_something_asynchronous()` would perform its asynchronous action and then it would resolve or reject the deferred, thereby concluding the promise and freeing the deferred:

```
napi_deferred deferred;
napi_value undefined;
napi_status status;

// Create a value with which to conclude the deferred.
status = napi_get_undefined(env, &undefined);
if (status != napi_ok) return NULL;

// Resolve or reject the promise associated with the deferred depending on
// whether the asynchronous action succeeded.
if (asynchronous_action_succeeded) {
  status = napi_resolve_deferred(env, deferred, undefined);
} else {
  status = napi_reject_deferred(env, deferred, undefined);
}
if (status != napi_ok) return NULL;

// At this point the deferred has been freed, so we should assign NULL to it.
deferred = NULL;
```

napi_create_promise

Added in: v8.5.0

```
napi_status napi_create_promise(napi_env env,
                                napi_deferred* deferred,
                                napi_value* promise);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] deferred` : A newly created deferred object which can later be passed to `napi_resolve_deferred()` or `napi_reject_deferred()` to resolve resp. reject the associated promise.
- `[out] promise` : The JavaScript promise associated with the deferred object.

Returns `napi_ok` if the API succeeded.

This API creates a deferred object and a JavaScript promise.

napi_resolve_deferred

Added in: v8.5.0

```
napi_status napi_resolve_deferred(napi_env env,
                                  napi_deferred deferred,
                                  napi_value resolution);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] deferred` : The deferred object whose associated promise to resolve.
- `[in] resolution` : The value with which to resolve the promise.

This API resolves a JavaScript promise by way of the deferred object with which it is associated. Thus, it can only be used to resolve JavaScript promises for which the corresponding deferred object is available. This effectively means that the promise must have been created using `napi_create_promise()` and the deferred object returned from that call must have been retained in order to be passed to this API.

The deferred object is freed upon successful completion.

napi_reject_deferred

Added in: v8.5.0

```
napi_status napi_reject_deferred(napi_env env,
                                 napi_deferred deferred,
                                 napi_value rejection);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] deferred` : The deferred object whose associated promise to resolve.
- `[in] rejection` : The value with which to reject the promise.

This API rejects a JavaScript promise by way of the deferred object with which it is associated. Thus, it can only be used to reject JavaScript promises for which the corresponding deferred object is available. This effectively means that the promise must have been created using `napi_create_promise()` and the deferred object returned from that call must have been retained in order to be passed to this API.

The deferred object is freed upon successful completion.

napi_is_promise

Added in: v8.5.0

```
napi_status napi_is_promise(napi_env env,
                            napi_value promise,
                            bool* is_promise);
```

- `[in] env` : The environment that the API is invoked under.

- `[in] promise` : The promise to examine
- `[out] is_promise` : Flag indicating whether `promise` is a native promise object - that is, a promise object created by the underlying engine.

Script execution

N-API provides an API for executing a string containing JavaScript using the underlying JavaScript engine.

napi_run_script

Added in: v8.5.0

```
NAPI_EXTERN napi_status napi_run_script(napi_env env,
                                        napi_value script,
                                        napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] script` : A JavaScript string containing the script to execute.
- `[out] result` : The value resulting from having executed the script.

libuv event loop

N-API provides a function for getting the current event loop associated with a specific `napi_env`.

napi_get_uv_event_loop

Added in: v9.3.0

```
NAPI_EXTERN napi_status napi_get_uv_event_loop(napi_env env,
                                               uv_loop_t** loop);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] loop` : The current libuv loop instance.

Child Process

Stability: 2 - Stable

The `child_process` module provides the ability to spawn child processes in a manner that is similar, but not identical, to popen(3) . This capability is primarily provided by the `child_process.spawn()` function:

```
const { spawn } = require('child_process');
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

By default, pipes for `stdin`, `stdout`, and `stderr` are established between the parent Node.js process and the spawned child. These pipes have limited (and platform-specific) capacity. If the child process writes to stdout in excess of that limit without the output being captured, the child process will block waiting for the pipe buffer to accept more data. This is identical to the behavior of pipes in the shell. Use the `{ stdio: 'ignore' }` option if the output will not be consumed.

The `child_process.spawn()` method spawns the child process asynchronously, without blocking the Node.js event loop. The `child_process.spawnSync()` function provides equivalent functionality in a synchronous manner that blocks the event loop until the spawned process either exits or is terminated.

For convenience, the `child_process` module provides a handful of synchronous and asynchronous alternatives to `child_process.spawn()` and `child_process.spawnSync()`. *Note that each of these alternatives are implemented on top of* `child_process.spawn()` *or* `child_process.spawnSync()`.

- `child_process.exec()`: spawns a shell and runs a command within that shell, passing the `stdout` and `stderr` to a callback function when complete.
- `child_process.execFile()`: similar to `child_process.exec()` except that it spawns the command directly without first spawning a shell by default.
- `child_process.fork()`: spawns a new Node.js process and invokes a specified module with an IPC communication channel established that allows sending messages between parent and child.
- `child_process.execSync()`: a synchronous version of `child_process.exec()` that *will* block the Node.js event loop.
- `child_process.execFileSync()`: a synchronous version of `child_process.execFile()` that *will* block the Node.js event loop.

For certain use cases, such as automating shell scripts, the synchronous counterparts may be more convenient. In many cases, however, the synchronous methods can have significant impact on performance due to stalling the event loop while spawned processes complete.

Asynchronous Process Creation

The `child_process.spawn()`, `child_process.fork()`, `child_process.exec()`, and `child_process.execFile()` methods all follow the idiomatic asynchronous programming pattern typical of other Node.js APIs.

Each of the methods returns a `ChildProcess` instance. These objects implement the Node.js `EventEmitter` API, allowing the parent process to register listener functions that are called when certain events occur during the life cycle of the child process.

The `child_process.exec()` and `child_process.execFile()` methods additionally allow for an optional `callback` function to be specified that is invoked when the child process terminates.

Spawning `.bat` and `.cmd` files on Windows

The importance of the distinction between `child_process.exec()` and `child_process.execFile()` can vary based on platform. On Unix-type operating systems (Unix, Linux, macOS) `child_process.execFile()` can be more efficient because it does not spawn a shell by default. On Windows, however, `.bat` and `.cmd` files are not executable on their own without a terminal, and therefore cannot be launched using `child_process.execFile()`. When running on Windows, `.bat` and `.cmd` files can be invoked using `child_process.spawn()` with the `shell` option set, with `child_process.exec()`, or by spawning `cmd.exe` and passing the `.bat` or `.cmd` file as an argument (which is what the `shell` option and `child_process.exec()` do). In any case, if the script filename contains spaces it needs to be quoted.

```js
// On Windows Only ...
const { spawn } = require('child_process');
const bat = spawn('cmd.exe', ['/c', 'my.bat']);

bat.stdout.on('data', (data) => {
  console.log(data.toString());
});

bat.stderr.on('data', (data) => {
  console.log(data.toString());
});

bat.on('exit', (code) => {
  console.log(`Child exited with code ${code}`);
});
```

```js
// OR...
const { exec } = require('child_process');
exec('my.bat', (err, stdout, stderr) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(stdout);
});

// Script with spaces in the filename:
const bat = spawn('"my script.cmd"', ['a', 'b'], { shell: true });
// or:
exec('"my script.cmd" a b', (err, stdout, stderr) => {
  // ...
});
```

child_process.exec(command[, options][, callback])

▸ History

- `command` <string> The command to run, with space-separated arguments.
- `options` <Object>
  - `cwd` <string> Current working directory of the child process.
  - `env` <Object> Environment key-value pairs.
  - `encoding` <string> **Default:** `'utf8'`
  - `shell` <string> Shell to execute the command with. **Default:** `'/bin/sh'` on UNIX, `process.env.ComSpec` on Windows. See Shell Requirements and Default Windows Shell.
  - `timeout` <number> **Default:** `0`
  - `maxBuffer` <number> Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated. See caveat at `maxBuffer` and Unicode. **Default:** `200 * 1024`.
  - `killSignal` <string> | <integer> **Default:** `'SIGTERM'`
  - `uid` <number> Sets the user identity of the process (see setuid(2)).
  - `gid` <number> Sets the group identity of the process (see setgid(2)).
  - `windowsHide` <boolean> Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- `callback` <Function> called with the output when process terminates.

    - `error` <Error>
    - `stdout` <string> | <Buffer>
    - `stderr` <string> | <Buffer>
- Returns: <ChildProcess>

Spawns a shell then executes the `command` within that shell, buffering any generated output. The `command` string passed to the exec function is processed directly by the shell and special characters (vary based on shell) need to be dealt with accordingly:

```
exec('"/path/to/test file/test.sh" arg1 arg2');
//Double quotes are used so that the space in the path is not interpreted as
//multiple arguments

exec('echo "The \\$HOME variable is $HOME"');
//The $HOME variable is escaped in the first instance, but not in the second
```

Never pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

```
const { exec } = require('child_process');
exec('cat *.js bad_file | wc -l', (error, stdout, stderr) => {
  if (error) {
    console.error(`exec error: ${error}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

If a `callback` function is provided, it is called with the arguments `(error, stdout, stderr)`. On success, `error` will be `null`. On error, `error` will be an instance of Error. The `error.code` property will be the exit code of the child process while `error.signal` will be set to the signal that terminated the process. Any exit code other than `0` is considered to be an error.

The `stdout` and `stderr` arguments passed to the callback will contain the stdout and stderr output of the child process. By default, Node.js will decode the output as UTF-8 and pass strings to the callback. The `encoding` option can be used to specify the character encoding used to decode the stdout and stderr output. If `encoding` is `'buffer'`, or an unrecognized character encoding, `Buffer` objects will be passed to the callback instead.

The `options` argument may be passed as the second argument to customize how the process is spawned. The default options are:

```
const defaults = {
  encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200 * 1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null
};
```

If `timeout` is greater than `0`, the parent will send the signal identified by the `killSignal` property (the default is `'SIGTERM'`) if the child runs longer than `timeout` milliseconds.

Unlike the exec(3) POSIX system call, `child_process.exec()` does not replace the existing process and uses a shell to execute the command.

If this method is invoked as its util.promisify()ed version, it returns a Promise for an object with `stdout` and `stderr` properties. In case of an error (including any error resulting in an exit code other than 0), a rejected promise is returned, with the same `error` object given in the callback, but with an additional two properties `stdout` and `stderr`.

```
const util = require('util');
const exec = util.promisify(require('child_process').exec);

async function lsExample() {
  const { stdout, stderr } = await exec('ls');
  console.log('stdout:', stdout);
  console.log('stderr:', stderr);
}
lsExample();
```

child_process.execFile(file[, args][, options][, callback])

▸ History

- `file` <string> The name or path of the executable file to run.
- `args` <string[]> List of string arguments.
- `options` <Object>
  - `cwd` <string> Current working directory of the child process.
  - `env` <Object> Environment key-value pairs.
  - `encoding` <string> **Default:** `'utf8'`
  - `timeout` <number> **Default:** `0`
  - `maxBuffer` <number> Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated. See caveat at `maxBuffer` and Unicode . **Default:** `200 * 1024`.
  - `killSignal` <string> | <integer> **Default:** `'SIGTERM'`
  - `uid` <number> Sets the user identity of the process (see setuid(2)).
  - `gid` <number> Sets the group identity of the process (see setgid(2)).
  - `windowsHide` <boolean> Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
  - `windowsVerbatimArguments` <boolean> No quoting or escaping of arguments is done on Windows. Ignored on Unix. **Default:** `false`.
  - `shell` <boolean> | <string> If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on UNIX, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See Shell Requirements and Default Windows Shell . **Default:** `false` (no shell).
- `callback` <Function> Called with the output when process terminates.
  - `error` <Error>
  - `stdout` <string> | <Buffer>
  - `stderr` <string> | <Buffer>
- Returns: <ChildProcess>

The `child_process.execFile()` function is similar to `child_process.exec()` except that it does not spawn a shell by default. Rather, the specified executable `file` is spawned directly as a new process making it slightly more efficient than `child_process.exec()`.

The same options as `child_process.exec()` are supported. Since a shell is not spawned, behaviors such as I/O redirection and file globbing are not supported.

```
const { execFile } = require('child_process');
const child = execFile('node', ['--version'], (error, stdout, stderr) => {
  if (error) {
    throw error;
  }
  console.log(stdout);
});
```

The `stdout` and `stderr` arguments passed to the callback will contain the stdout and stderr output of the child process. By default, Node.js will decode the output as UTF-8 and pass strings to the callback. The `encoding` option can be used to specify the character encoding used to

decode the stdout and stderr output. If `encoding` is `'buffer'`, or an unrecognized character encoding, `Buffer` objects will be passed to the callback instead.

If this method is invoked as its util.promisify() ed version, it returns a Promise for an object with `stdout` and `stderr` properties. In case of an error (including any error resulting in an exit code other than 0), a rejected promise is returned, with the same `error` object given in the callback, but with an additional two properties `stdout` and `stderr`.

```
const util = require('util');
const execFile = util.promisify(require('child_process').execFile);
async function getVersion() {
  const { stdout } = await execFile('node', ['--version']);
  console.log(stdout);
}
getVersion();
```

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

child_process.fork(modulePath[, args][, options])

▸ History

- `modulePath` <string> The module to run in the child.
- `args` <Array> List of string arguments.
- `options` <Object>
  - `cwd` <string> Current working directory of the child process.
  - `env` <Object> Environment key-value pairs.
  - `execPath` <string> Executable used to create the child process.
  - `execArgv` <Array> List of string arguments passed to the executable. **Default:** `process.execArgv`
  - `silent` <boolean> If `true`, stdin, stdout, and stderr of the child will be piped to the parent, otherwise they will be inherited from the parent, see the `'pipe'` and `'inherit'` options for child_process.spawn() 's stdio for more details. **Default:** `false`
  - `stdio` <Array> | <string> See child_process.spawn() 's stdio . When this option is provided, it overrides `silent`. If the array variant is used, it must contain exactly one item with value `'ipc'` or an error will be thrown. For instance `[0, 1, 2, 'ipc']`.
  - `windowsVerbatimArguments` <boolean> No quoting or escaping of arguments is done on Windows. Ignored on Unix. **Default:** `false`.
  - `uid` <number> Sets the user identity of the process (see setuid(2)).
  - `gid` <number> Sets the group identity of the process (see setgid(2)).
- Returns: <ChildProcess>

The `child_process.fork()` method is a special case of child_process.spawn() used specifically to spawn new Node.js processes. Like child_process.spawn() , a ChildProcess object is returned. The returned ChildProcess will have an additional communication channel built-in that allows messages to be passed back and forth between the parent and child. See subprocess.send() for details.

It is important to keep in mind that spawned Node.js child processes are independent of the parent with exception of the IPC communication channel that is established between the two. Each process has its own memory, with their own V8 instances. Because of the additional resource allocations required, spawning a large number of child Node.js processes is not recommended.

By default, `child_process.fork()` will spawn new Node.js instances using the process.execPath of the parent process. The `execPath` property in the `options` object allows for an alternative execution path to be used.

Node.js processes launched with a custom `execPath` will communicate with the parent process using the file descriptor (fd) identified using the environment variable `NODE_CHANNEL_FD` on the child process.

Unlike the fork(2) POSIX system call, `child_process.fork()` does not clone the current process.

The `shell` option available in child_process.spawn() is not supported by `child_process.fork()` and will be ignored if set.

child_process.spawn(command[, args][, options])

▸ History

- `command` <string> The command to run.
- `args` <Array> List of string arguments.
- `options` <Object>
  - `cwd` <string> Current working directory of the child process.
  - `env` <Object> Environment key-value pairs.
  - `argv0` <string> Explicitly set the value of `argv[0]` sent to the child process. This will be set to `command` if not specified.
  - `stdio` <Array> | <string> Child's stdio configuration (see `options.stdio`).
  - `detached` <boolean> Prepare child to run independently of its parent process. Specific behavior depends on the platform, see `options.detached`).
  - `uid` <number> Sets the user identity of the process (see setuid(2)).
  - `gid` <number> Sets the group identity of the process (see setgid(2)).
  - `shell` <boolean> | <string> If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on UNIX, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See Shell Requirements and Default Windows Shell. **Default:** `false` (no shell).
  - `windowsVerbatimArguments` <boolean> No quoting or escaping of arguments is done on Windows. Ignored on Unix. This is set to `true` automatically when `shell` is specified. **Default:** `false`.
  - `windowsHide` <boolean> Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- Returns: <ChildProcess>

The `child_process.spawn()` method spawns a new process using the given `command`, with command line arguments in `args`. If omitted, `args` defaults to an empty array.

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

A third argument may be used to specify additional options, with these defaults:

```
const defaults = {
  cwd: undefined,
  env: process.env
};
```

Use `cwd` to specify the working directory from which the process is spawned. If not given, the default is to inherit the current working directory.

Use `env` to specify environment variables that will be visible to the new process, the default is `process.env`.

Example of running `ls -lh /usr`, capturing `stdout`, `stderr`, and the exit code:

```
const { spawn } = require('child_process');
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

Example: A very elaborate way to run `ps ax | grep ssh`

```
const { spawn } = require('child_process');
const ps = spawn('ps', ['ax']);
const grep = spawn('grep', ['ssh']);

ps.stdout.on('data', (data) => {
  grep.stdin.write(data);
});

ps.stderr.on('data', (data) => {
  console.log(`ps stderr: ${data}`);
});

ps.on('close', (code) => {
  if (code !== 0) {
    console.log(`ps process exited with code ${code}`);
  }
  grep.stdin.end();
});

grep.stdout.on('data', (data) => {
  console.log(data.toString());
});

grep.stderr.on('data', (data) => {
  console.log(`grep stderr: ${data}`);
});

grep.on('close', (code) => {
  if (code !== 0) {
    console.log(`grep process exited with code ${code}`);
  }
});
```

Example of checking for failed `spawn`:

```
const { spawn } = require('child_process');
const subprocess = spawn('bad_command');

subprocess.on('error', (err) => {
  console.log('Failed to start subprocess.');
});
```

Certain platforms (macOS, Linux) will use the value of `argv[0]` for the process title while others (Windows, SunOS) will use `command`.

Node.js currently overwrites `argv[0]` with `process.execPath` on startup, so `process.argv[0]` in a Node.js child process will not match the `argv0` parameter passed to `spawn` from the parent, retrieve it with the `process.argv0` property instead.

options.detached

Added in: v0.7.10

On Windows, setting `options.detached` to `true` makes it possible for the child process to continue running after the parent exits. The child will have its own console window. *Once enabled for a child process, it cannot be disabled.*

On non-Windows platforms, if `options.detached` is set to `true`, the child process will be made the leader of a new process group and session. Note that child processes may continue running after the parent exits regardless of whether they are detached or not. See setsid(2) for more information.

By default, the parent will wait for the detached child to exit. To prevent the parent from waiting for a given `subprocess`, use the `subprocess.unref()` method. Doing so will cause the parent's event loop to not include the child in its reference count, allowing the parent to exit independently of the child, unless there is an established IPC channel between the child and parent.

When using the `detached` option to start a long-running process, the process will not stay running in the background after the parent exits unless it is provided with a `stdio` configuration that is not connected to the parent. If the parent's `stdio` is inherited, the child will remain attached to the controlling terminal.

Example of a long-running process, by detaching and also ignoring its parent `stdio` file descriptors, in order to ignore the parent's termination:

```
const { spawn } = require('child_process');

const subprocess = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: 'ignore'
});

subprocess.unref();
```

Alternatively one can redirect the child process' output into files:

```
const fs = require('fs');
const { spawn } = require('child_process');
const out = fs.openSync('./out.log', 'a');
const err = fs.openSync('./out.log', 'a');

const subprocess = spawn('prg', [], {
  detached: true,
  stdio: [ 'ignore', out, err ]
});

subprocess.unref();
```

options.stdio

▸ History

The `options.stdio` option is used to configure the pipes that are established between the parent and child process. By default, the child's stdin, stdout, and stderr are redirected to corresponding `subprocess.stdin`, `subprocess.stdout`, and `subprocess.stderr` streams on the `ChildProcess` object. This is equivalent to setting the `options.stdio` equal to `['pipe', 'pipe', 'pipe']`.

For convenience, `options.stdio` may be one of the following strings:

- `'pipe'` - equivalent to `['pipe', 'pipe', 'pipe']` (the default)
- `'ignore'` - equivalent to `['ignore', 'ignore', 'ignore']`
- `'inherit'` - equivalent to `[process.stdin, process.stdout, process.stderr]` or `[0,1,2]`

Otherwise, the value of `options.stdio` is an array where each index corresponds to an fd in the child. The fds 0, 1, and 2 correspond to stdin, stdout, and stderr, respectively. Additional fds can be specified to create additional pipes between the parent and child. The value is one of the following:

1. `'pipe'` - Create a pipe between the child process and the parent process. The parent end of the pipe is exposed to the parent as a property on the `child_process` object as `subprocess.stdio[fd]`. Pipes created for fds 0 - 2 are also available as `subprocess.stdin`, `subprocess.stdout` and `subprocess.stderr`, respectively.
2. `'ipc'` - Create an IPC channel for passing messages/file descriptors between parent and child. A `ChildProcess` may have at most *one* IPC stdio file descriptor. Setting this option enables the `subprocess.send()` method. If the child is a Node.js process, the presence of an IPC channel will enable `process.send()`, `process.disconnect()`, `process.on('disconnect')`, and `process.on('message')` within the child.

   Accessing the IPC channel fd in any way other than `process.send()` or using the IPC channel with a child process that is not a Node.js instance is not supported.

3. `'ignore'` - Instructs Node.js to ignore the fd in the child. While Node.js will always open fds 0 - 2 for the processes it spawns, setting the fd to `'ignore'` will cause Node.js to open `/dev/null` and attach it to the child's fd.
4. <Stream> object - Share a readable or writable stream that refers to a tty, file, socket, or a pipe with the child process. The stream's underlying file descriptor is duplicated in the child process to the fd that corresponds to the index in the `stdio` array. Note that the stream must have an underlying descriptor (file streams do not until the `'open'` event has occurred).
5. Positive integer - The integer value is interpreted as a file descriptor that is currently open in the parent process. It is shared with the child process, similar to how <Stream> objects can be shared.
6. `null`, `undefined` - Use default value. For stdio fds 0, 1, and 2 (in other words, stdin, stdout, and stderr) a pipe is created. For fd 3 and up, the default is `'ignore'`.

Example:

```
const { spawn } = require('child_process');

// Child will use parent's stdios
spawn('prg', [], { stdio: 'inherit' });

// Spawn child sharing only stderr
spawn('prg', [], { stdio: ['pipe', 'pipe', process.stderr] });

// Open an extra fd=4, to interact with programs presenting a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe'] });
```

It is worth noting that when an IPC channel is established between the parent and child processes, and the child is a Node.js process, the child is launched with the IPC channel unreferenced (using `unref()`) until the child registers an event handler for the `process.on('disconnect')` event or the `process.on('message')` event. This allows the child to exit normally without the process being held open by the open IPC channel.

See also: `child_process.exec()` and `child_process.fork()`

Synchronous Process Creation

The `child_process.spawnSync()`, `child_process.execSync()`, and `child_process.execFileSync()` methods are **synchronous** and **WILL** block the

Node.js event loop, pausing execution of any additional code until the spawned process exits.

Blocking calls like these are mostly useful for simplifying general-purpose scripting tasks and for simplifying the loading/processing of application configuration at startup.

child_process.execFileSync(file[, args][, options])

▸ History

- `file` <string> The name or path of the executable file to run.
- `args` <string[]> List of string arguments.
- `options` <Object>
  - `cwd` <string> Current working directory of the child process.
  - `input` <string> | <Buffer> | <Uint8Array> The value which will be passed as stdin to the spawned process.
    - supplying this value will override `stdio[0]`
  - `stdio` <string> | <Array> Child's stdio configuration. **Default:** `'pipe'`
    - `stderr` by default will be output to the parent process' stderr unless `stdio` is specified
  - `env` <Object> Environment key-value pairs.
  - `uid` <number> Sets the user identity of the process (see setuid(2)).
  - `gid` <number> Sets the group identity of the process (see setgid(2)).
  - `timeout` <number> In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`
  - `killSignal` <string> | <integer> The signal value to be used when the spawned process will be killed.**Default:** `'SIGTERM'`
  - `maxBuffer` <number> Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated. See caveat at maxBuffer and Unicode . **Default:** `200 * 1024` .
  - `encoding` <string> The encoding used for all stdio inputs and outputs. **Default:** `'buffer'` .
  - `windowsHide` <boolean> Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false` .
  - `shell` <boolean> | <string> If `true` , runs `command` inside of a shell. Uses `'/bin/sh'` on UNIX, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See Shell Requirements and Default Windows Shell . **Default:** `false` (no shell).
- Returns: <Buffer> | <string> The stdout from the command.

The `child_process.execFileSync()` method is generally identical to `child_process.execFile()` with the exception that the method will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely exited.

If the child process intercepts and handles the `SIGTERM` signal and does not exit, the parent process will still wait until the child process has exited.

If the process times out or has a non-zero exit code, this method***will*** throw an `Error` that will include the full result of the underlying `child_process.spawnSync()` .

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

child_process.execSync(command[, options])

▸ History

- `command` <string> The command to run.
- `options` <Object>
  - `cwd` <string> Current working directory of the child process.
  - `input` <string> | <Buffer> | <Uint8Array> The value which will be passed as stdin to the spawned process.
    - supplying this value will override `stdio[0]` .
  - `stdio` <string> | <Array> Child's stdio configuration. **Default:** `'pipe'`
    - `stderr` by default will be output to the parent process' stderr unless `stdio` is specified
  - `env` <Object> Environment key-value pairs.

- shell <string> Shell to execute the command with. **Default:** '/bin/sh' on UNIX, process.env.ComSpec on Windows. See Shell Requirements and Default Windows Shell .
- uid <number> Sets the user identity of the process. (See setuid(2)).
- gid <number> Sets the group identity of the process. (See setgid(2)).
- timeout <number> In milliseconds the maximum amount of time the process is allowed to run. **Default:** undefined
- killSignal <string> | <integer> The signal value to be used when the spawned process will be killed. **Default:** 'SIGTERM'
- maxBuffer <number> Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated. See caveat at maxBuffer and Unicode . **Default:** 200 * 1024 .
- encoding <string> The encoding used for all stdio inputs and outputs. **Default:** 'buffer'
- windowsHide <boolean> Hide the subprocess console window that would normally be created on Windows systems. **Default:** false .

* Returns: <Buffer> | <string> The stdout from the command.

The child_process.execSync() method is generally identical to child_process.exec() with the exception that the method will not return until the child process has fully closed. When a timeout has been encountered and killSignal is sent, the method won't return until the process has completely exited. *Note that if the child process intercepts and handles the* SIGTERM *signal and doesn't exit, the parent process will wait until the child process has exited.*

If the process times out or has a non-zero exit code, this method ***will*** throw. The Error object will contain the entire result from child_process.spawnSync()

Never pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

child_process.spawnSync(command[, args][, options])

▸ History

* command <string> The command to run.
* args <Array> List of string arguments.
* options <Object>
  - cwd <string> Current working directory of the child process.
  - input <string> | <Buffer> | <Uint8Array> The value which will be passed as stdin to the spawned process.
    - supplying this value will override stdio[0] .
  - stdio <string> | <Array> Child's stdio configuration.
  - env <Object> Environment key-value pairs.
  - uid <number> Sets the user identity of the process (see setuid(2)).
  - gid <number> Sets the group identity of the process (see setgid(2)).
  - timeout <number> In milliseconds the maximum amount of time the process is allowed to run. **Default:** undefined
  - killSignal <string> | <integer> The signal value to be used when the spawned process will be killed. **Default:** 'SIGTERM'
  - maxBuffer <number> Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated. See caveat at maxBuffer and Unicode . **Default:** 200 * 1024 .
  - encoding <string> The encoding used for all stdio inputs and outputs. **Default:** 'buffer'
  - shell <boolean> | <string> If true , runs command inside of a shell. Uses '/bin/sh' on UNIX, and process.env.ComSpec on Windows. A different shell can be specified as a string. See Shell Requirements and Default Windows Shell . **Default:** false (no shell).
  - windowsVerbatimArguments <boolean> No quoting or escaping of arguments is done on Windows. Ignored on Unix. This is set to true automatically when shell is specified. **Default:** false .
  - windowsHide <boolean> Hide the subprocess console window that would normally be created on Windows systems. **Default:** false .
* Returns: <Object>
  - pid <number> Pid of the child process.
  - output <Array> Array of results from stdio output.
  - stdout <Buffer> | <string> The contents of output[1] .
  - stderr <Buffer> | <string> The contents of output[2] .

- `status` <number> The exit code of the child process.
- `signal` <string> The signal used to kill the child process.
- `error` <Error> The error object if the child process failed or timed out.

The `child_process.spawnSync()` method is generally identical to `child_process.spawn()` with the exception that the function will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely exited. Note that if the process intercepts and handles the `SIGTERM` signal and doesn't exit, the parent process will wait until the child process has exited.

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

Class: ChildProcess

Added in: v2.2.0

Instances of the `ChildProcess` class are `EventEmitters` that represent spawned child processes.

Instances of `ChildProcess` are not intended to be created directly. Rather, use the `child_process.spawn()`, `child_process.exec()`, `child_process.execFile()`, or `child_process.fork()` methods to create instances of `ChildProcess`.

Event: 'close'

Added in: v0.7.7

- `code` <number> The exit code if the child exited on its own.
- `signal` <string> The signal by which the child process was terminated.

The `'close'` event is emitted when the stdio streams of a child process have been closed. This is distinct from the `'exit'` event, since multiple processes might share the same stdio streams.

Event: 'disconnect'

Added in: v0.7.2

The `'disconnect'` event is emitted after calling the `subprocess.disconnect()` method in parent process or `process.disconnect()` in child process. After disconnecting it is no longer possible to send or receive messages, and the `subprocess.connected` property is `false`.

Event: 'error'

- `err` <Error> The error.

The `'error'` event is emitted whenever:

1. The process could not be spawned, or
2. The process could not be killed, or
3. Sending a message to the child process failed.

The `'exit'` event may or may not fire after an error has occurred. When listening to both the `'exit'` and `'error'` events, it is important to guard against accidentally invoking handler functions multiple times.

See also `subprocess.kill()` and `subprocess.send()`.

Event: 'exit'

Added in: v0.1.90

- `code` <number> The exit code if the child exited on its own.
- `signal` <string> The signal by which the child process was terminated.

The `'exit'` event is emitted after the child process ends. If the process exited, `code` is the final exit code of the process, otherwise `null`. If the process terminated due to receipt of a signal, `signal` is the string name of the signal, otherwise `null`. One of the two will always be non-null.

Note that when the `'exit'` event is triggered, child process stdio streams might still be open.

Also, note that Node.js establishes signal handlers for `SIGINT` and `SIGTERM` and Node.js processes will not terminate immediately due to receipt of those signals. Rather, Node.js will perform a sequence of cleanup actions and then will re-raise the handled signal.

See waitpid(2) .

Event: 'message'

Added in: v0.5.9

- `message` <Object> A parsed JSON object or primitive value.
- `sendHandle` <Handle> A net.Socket or net.Server object, or undefined.

The `'message'` event is triggered when a child process uses process.send() to send messages.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

subprocess.channel

Added in: v7.1.0

- <Object> A pipe representing the IPC channel to the child process.

The `subprocess.channel` property is a reference to the child's IPC channel. If no IPC channel currently exists, this property is `undefined`.

subprocess.connected

Added in: v0.7.2

- <boolean> Set to `false` after `subprocess.disconnect()` is called.

The `subprocess.connected` property indicates whether it is still possible to send and receive messages from a child process. When `subprocess.connected` is `false`, it is no longer possible to send or receive messages.

subprocess.disconnect()

Added in: v0.7.2

Closes the IPC channel between parent and child, allowing the child to exit gracefully once there are no other connections keeping it alive. After calling this method the `subprocess.connected` and `process.connected` properties in both the parent and child (respectively) will be set to `false`, and it will be no longer possible to pass messages between the processes.

The `'disconnect'` event will be emitted when there are no messages in the process of being received. This will most often be triggered immediately after calling `subprocess.disconnect()`.

Note that when the child process is a Node.js instance (e.g. spawned using child_process.fork()), the `process.disconnect()` method can be invoked within the child process to close the IPC channel as well.

subprocess.kill([signal])

Added in: v0.1.90

- `signal` <string>

The `subprocess.kill()` method sends a signal to the child process. If no argument is given, the process will be sent the `'SIGTERM'` signal. See signal(7) for a list of available signals.

```js
const { spawn } = require('child_process');
const grep = spawn('grep', ['ssh']);

grep.on('close', (code, signal) => {
  console.log(
    `child process terminated due to receipt of signal ${signal}`);
});

// Send SIGHUP to process
grep.kill('SIGHUP');
```

The `ChildProcess` object may emit an `'error'` event if the signal cannot be delivered. Sending a signal to a child process that has already exited is not an error but may have unforeseen consequences. Specifically, if the process identifier (PID) has been reassigned to another process, the signal will be delivered to that process instead which can have unexpected results.

Note that while the function is called `kill`, the signal delivered to the child process may not actually terminate the process.

See kill(2) for reference.

Also note: on Linux, child processes of child processes will not be terminated when attempting to kill their parent. This is likely to happen when running a new process in a shell or with use of the `shell` option of `ChildProcess`, such as in this example:

```js
'use strict';
const { spawn } = require('child_process');

const subprocess = spawn(
  'sh',
  [
    '-c',
    `node -e "setInterval(() => {
      console.log(process.pid, 'is alive')
    }, 500);"`
  ], {
    stdio: ['inherit', 'inherit', 'inherit']
  }
);

setTimeout(() => {
  subprocess.kill(); // does not terminate the node process in the shell
}, 2000);
```

subprocess.killed

Added in: v0.5.10

- <boolean> Set to `true` after `subprocess.kill()` is used to successfully send a signal to the child process.

The `subprocess.killed` property indicates whether the child process successfully received a signal from `subprocess.kill()`. The `killed` property does not indicate that the child process has been terminated.

subprocess.pid

Added in: v0.1.90

- <number> Integer

Returns the process identifier (PID) of the child process.

Example:

```
const { spawn } = require('child_process');
const grep = spawn('grep', ['ssh']);

console.log(`Spawned child pid: ${grep.pid}`);
grep.stdin.end();
```

subprocess.send(message[, sendHandle[, options]][, callback])

▸ History

- `message` <Object>
- `sendHandle` <Handle>
- `options` <Object>
- `callback` <Function>
- Returns: <boolean>

When an IPC channel has been established between the parent and child (i.e. when using child_process.fork()), the `subprocess.send()` method can be used to send messages to the child process. When the child process is a Node.js instance, these messages can be received via the process.on('message') event.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

For example, in the parent script:

```
const cp = require('child_process');
const n = cp.fork(`${__dirname}/sub.js`);

n.on('message', (m) => {
  console.log('PARENT got message:', m);
});

// Causes the child to print: CHILD got message: { hello: 'world' }
n.send({ hello: 'world' });
```

And then the child script, `'sub.js'` might look like this:

```
process.on('message', (m) => {
  console.log('CHILD got message:', m);
});

// Causes the parent to print: PARENT got message: { foo: 'bar', baz: null }
process.send({ foo: 'bar', baz: NaN });
```

Child Node.js processes will have a process.send() method of their own that allows the child to send messages back to the parent.

There is a special case when sending a `{cmd: 'NODE_foo'}` message. Messages containing a `NODE_` prefix in the `cmd` property are reserved for use within Node.js core and will not be emitted in the child's process.on('message') event. Rather, such messages are emitted using the `process.on('internalMessage')` event and are consumed internally by Node.js. Applications should avoid using such messages or listening for `'internalMessage'` events as it is subject to change without notice.

The optional `sendHandle` argument that may be passed to `subprocess.send()` is for passing a TCP server or socket object to the child process. The child will receive the object as the second argument passed to the callback function registered on the process.on('message') event. Any data that is received and buffered in the socket will not be sent to the child.

The `options` argument, if present, is an object used to parameterize the sending of certain types of handles. `options` supports the following properties:

- `keepOpen` - A Boolean value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. Defaults to `false`.

The optional `callback` is a function that is invoked after the message is sent but before the child may have received it. The function is called with a single argument: `null` on success, or an `Error` object on failure.

If no `callback` function is provided and the message cannot be sent, an `'error'` event will be emitted by the `ChildProcess` object. This can happen, for instance, when the child process has already exited.

`subprocess.send()` will return `false` if the channel has closed or when the backlog of unsent messages exceeds a threshold that makes it unwise to send more. Otherwise, the method returns `true`. The `callback` function can be used to implement flow control.

Example: sending a server object

The `sendHandle` argument can be used, for instance, to pass the handle of a TCP server object to the child process as illustrated in the example below:

```
const subprocess = require('child_process').fork('subprocess.js');

// Open up the server object and send the handle.
const server = require('net').createServer();
server.on('connection', (socket) => {
  socket.end('handled by parent');
});
server.listen(1337, () => {
  subprocess.send('server', server);
});
```

The child would then receive the server object as:

```
process.on('message', (m, server) => {
  if (m === 'server') {
    server.on('connection', (socket) => {
      socket.end('handled by child');
    });
  }
});
```

Once the server is now shared between the parent and child, some connections can be handled by the parent and some by the child.

While the example above uses a server created using the `net` module, `dgram` module servers use exactly the same workflow with the exceptions of listening on a `'message'` event instead of `'connection'` and using `server.bind()` instead of `server.listen()`. This is, however, currently only supported on UNIX platforms.

Example: sending a socket object

Similarly, the `sendHandler` argument can be used to pass the handle of a socket to the child process. The example below spawns two children that each handle connections with "normal" or "special" priority:

```
const { fork } = require('child_process');
const normal = fork('subprocess.js', ['normal']);
const special = fork('subprocess.js', ['special']);

// Open up the server and send sockets to child. Use pauseOnConnect to prevent
// the sockets from being read before they are sent to the child process.
const server = require('net').createServer({ pauseOnConnect: true });
server.on('connection', (socket) => {

  // If this is special priority
  if (socket.remoteAddress === '74.125.127.100') {
    special.send('socket', socket);
    return;
  }
  // This is normal priority
  normal.send('socket', socket);
});
server.listen(1337);
```

The `subprocess.js` would receive the socket handle as the second argument passed to the event callback function:

```
process.on('message', (m, socket) => {
  if (m === 'socket') {
    if (socket) {
      // Check that the client socket exists.
      // It is possible for the socket to be closed between the time it is
      // sent and the time it is received in the child process.
      socket.end(`Request handled with ${process.argv[2]} priority`);
    }
  }
});
```

Once a socket has been passed to a child, the parent is no longer capable of tracking when the socket is destroyed. To indicate this, the `.connections` property becomes `null`. It is recommended not to use `.maxConnections` when this occurs.

It is also recommended that any `'message'` handlers in the child process verify that `socket` exists, as the connection may have been closed during the time it takes to send the connection to the child.

subprocess.stderr

Added in: v0.1.90

- <stream.Readable>

A `Readable Stream` that represents the child process's `stderr`.

If the child was spawned with `stdio[2]` set to anything other than `'pipe'`, then this will be `null`.

`subprocess.stderr` is an alias for `subprocess.stdio[2]`. Both properties will refer to the same value.

subprocess.stdin

Added in: v0.1.90

- <stream.Writable>

A `Writable Stream` that represents the child process's `stdin`.

Note that if a child process waits to read all of its input, the child will not continue until this stream has been closed via `end()`.

If the child was spawned with `stdio[0]` set to anything other than `'pipe'`, then this will be `null`.

`subprocess.stdin` is an alias for `subprocess.stdio[0]`. Both properties will refer to the same value.

subprocess.stdio

Added in: v0.7.10

- <Array>

A sparse array of pipes to the child process, corresponding with positions in the `stdio` option passed to `child_process.spawn()` that have been set to the value `'pipe'`. Note that `subprocess.stdio[0]`, `subprocess.stdio[1]`, and `subprocess.stdio[2]` are also available as `subprocess.stdin`, `subprocess.stdout`, and `subprocess.stderr`, respectively.

In the following example, only the child's fd `1` (stdout) is configured as a pipe, so only the parent's `subprocess.stdio[1]` is a stream, all other values in the array are `null`.

```
const assert = require('assert');
const fs = require('fs');
const child_process = require('child_process');

const subprocess = child_process.spawn('ls', {
  stdio: [
    0, // Use parent's stdin for child
    'pipe', // Pipe child's stdout to parent
    fs.openSync('err.out', 'w') // Direct child's stderr to a file
  ]
});

assert.strictEqual(subprocess.stdio[0], null);
assert.strictEqual(subprocess.stdio[0], subprocess.stdin);

assert(subprocess.stdout);
assert.strictEqual(subprocess.stdio[1], subprocess.stdout);

assert.strictEqual(subprocess.stdio[2], null);
assert.strictEqual(subprocess.stdio[2], subprocess.stderr);
```

subprocess.stdout

Added in: v0.1.90

- <stream.Readable>

A `Readable Stream` that represents the child process's `stdout`.

If the child was spawned with `stdio[1]` set to anything other than `'pipe'`, then this will be `null`.

`subprocess.stdout` is an alias for `subprocess.stdio[1]`. Both properties will refer to the same value.

`maxBuffer` and Unicode

The `maxBuffer` option specifies the largest number of bytes allowed on `stdout` or `stderr`. If this value is exceeded, then the child process is terminated. This impacts output that includes multibyte character encodings such as UTF-8 or UTF-16. For instance, `console.log('□□□□')` will send 13 UTF-8 encoded bytes to `stdout` although there are only 4 characters.

Shell Requirements

The shell should understand the `-c` switch on UNIX or `/d /s /c` on Windows. On Windows, command line parsing should be compatible with `'cmd.exe'`.

Default Windows Shell

Although Microsoft specifies `%COMSPEC%` must contain the path to `'cmd.exe'` in the root environment, child processes are not always subject to the same requirement. Thus, in `child_process` functions where a shell can be spawned, `'cmd.exe'` is used as a fallback if `process.env.ComSpec` is unavailable.

Cluster

Stability: 2 - Stable

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems, the user will sometimes want to launch a cluster of Node.js processes to handle the load.

The cluster module allows easy creation of child processes that all share server ports.

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);

  console.log(`Worker ${process.pid} started`);
}
```

Running Node.js will now share port 8000 between the workers:

```
$ node server.js
Master 3596 is running
Worker 4324 started
Worker 4520 started
Worker 6056 started
Worker 5644 started
```

Please note that on Windows, it is not yet possible to set up a named pipe server in a worker.

How It Works

The worker processes are spawned using the `child_process.fork()` method, so that they can communicate with the parent via IPC and pass

server handles back and forth.

The cluster module supports two methods of distributing incoming connections.

The first one (and the default one on all platforms except Windows), is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process.

The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.

The second approach should, in theory, give the best performance. In practice however, distribution tends to be very unbalanced due to operating system scheduler vagaries. Loads have been observed where over 70% of all connections ended up in just two processes, out of a total of eight.

Because `server.listen()` hands off most of the work to the master process, there are three cases where the behavior between a normal Node.js process and a cluster worker differs:

1. `server.listen({fd: 7})` Because the message is passed to the master, file descriptor 7 **in the parent** will be listened on, and the handle passed to the worker, rather than listening to the worker's idea of what the number 7 file descriptor references.
2. `server.listen(handle)` Listening on handles explicitly will cause the worker to use the supplied handle, rather than talk to the master process.
3. `server.listen(0)` Normally, this will cause servers to listen on a random port. However, in a cluster, each worker will receive the same "random" port each time they do `listen(0)`. In essence, the port is random the first time, but predictable thereafter. To listen on a unique port, generate a port number based on the cluster worker ID.

Node.js does not provide routing logic. It is, therefore important to design an application such that it does not rely too heavily on in-memory data objects for things like sessions and login.

Because workers are all separate processes, they can be killed or re-spawned depending on a program's needs, without affecting other workers. As long as there are some workers still alive, the server will continue to accept connections. If no workers are alive, existing connections will be dropped and new connections will be refused. Node.js does not automatically manage the number of workers, however. It is the application's responsibility to manage the worker pool based on its own needs.

Although a primary use case for the `cluster` module is networking, it can also be used for other use cases requiring worker processes.

Class: Worker

Added in: v0.7.0

A Worker object contains all public information and method about a worker. In the master it can be obtained using `cluster.workers`. In a worker it can be obtained using `cluster.worker`.

Event: 'disconnect'

Added in: v0.7.7

Similar to the `cluster.on('disconnect')` event, but specific to this worker.

```
cluster.fork().on('disconnect', () => {
  // Worker has disconnected
});
```

Event: 'error'

Added in: v0.7.3

This event is the same as the one provided by `child_process.fork()`.

Within a worker, `process.on('error')` may also be used.

Event: 'exit'

Added in: v0.11.2

- `code` <number> The exit code, if it exited normally.
- `signal` <string> The name of the signal (e.g. `'SIGHUP'`) that caused the process to be killed.

Similar to the `cluster.on('exit')` event, but specific to this worker.

```
const worker = cluster.fork();
worker.on('exit', (code, signal) => {
  if (signal) {
    console.log(`worker was killed by signal: ${signal}`);
  } else if (code !== 0) {
    console.log(`worker exited with error code: ${code}`);
  } else {
    console.log('worker success!');
  }
});
```

Event: 'listening'

Added in: v0.7.0

- `address` <Object>

Similar to the `cluster.on('listening')` event, but specific to this worker.

```
cluster.fork().on('listening', (address) => {
  // Worker is listening
});
```

It is not emitted in the worker.

Event: 'message'

Added in: v0.7.0

- `message` <Object>
- `handle` <undefined> | <Object>

Similar to the `cluster.on('message')` event, but specific to this worker.

Within a worker, `process.on('message')` may also be used.

See process event: `'message'`.

As an example, here is a cluster that keeps count of the number of requests in the master process using the message system:

```js
const cluster = require('cluster');
const http = require('http');

if (cluster.isMaster) {

  // Keep track of http requests
  let numReqs = 0;
  setInterval(() => {
    console.log(`numReqs = ${numReqs}`);
  }, 1000);

  // Count requests
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd === 'notifyRequest') {
      numReqs += 1;
    }
  }

  // Start workers and listen for messages containing notifyRequest
  const numCPUs = require('os').cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  for (const id in cluster.workers) {
    cluster.workers[id].on('message', messageHandler);
  }

} else {

  // Worker processes have a http server.
  http.Server((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');

    // notify master about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}
```

Event: 'online'

Added in: v0.7.0

Similar to the `cluster.on('online')` event, but specific to this worker.

```js
cluster.fork().on('online', () => {
  // Worker is online
});
```

It is not emitted in the worker.

worker.disconnect()

▸ History

- Returns: <cluster.Worker> A reference to `worker`.

In a worker, this function will close all servers, wait for the `'close'` event on those servers, and then disconnect the IPC channel.

In the master, an internal message is sent to the worker causing it to call `.disconnect()` on itself.

Causes `.exitedAfterDisconnect` to be set.

Note that after a server is closed, it will no longer accept new connections, but connections may be accepted by any other listening worker. Existing connections will be allowed to close as usual. When no more connections exist, see server.close(), the IPC channel to the worker will close allowing it to die gracefully.

The above applies *only* to server connections, client connections are not automatically closed by workers, and disconnect does not wait for them to close before exiting.

Note that in a worker, `process.disconnect` exists, but it is not this function, it is disconnect.

Because long living server connections may block workers from disconnecting, it may be useful to send a message, so application specific actions may be taken to close them. It also may be useful to implement a timeout, killing a worker if the `'disconnect'` event has not been emitted after some time.

```
if (cluster.isMaster) {
  const worker = cluster.fork();
  let timeout;

  worker.on('listening', (address) => {
    worker.send('shutdown');
    worker.disconnect();
    timeout = setTimeout(() => {
      worker.kill();
    }, 2000);
  });

  worker.on('disconnect', () => {
    clearTimeout(timeout);
  });

} else if (cluster.isWorker) {
  const net = require('net');
  const server = net.createServer((socket) => {
    // connections never end
  });

  server.listen(8000);

  process.on('message', (msg) => {
    if (msg === 'shutdown') {
      // initiate graceful close of any connections to server
    }
  });
}
```

worker.exitedAfterDisconnect

Added in: v6.0.0

- <boolean>

Set by calling `.kill()` or `.disconnect()`. Until then, it is `undefined`.

The boolean worker.exitedAfterDisconnect allows distinguishing between voluntary and accidental exit, the master may choose not to respawn a worker based on this value.

```
cluster.on('exit', (worker, code, signal) => {
  if (worker.exitedAfterDisconnect === true) {
    console.log('Oh, it was just voluntary – no need to worry');
  }
});

// kill worker
worker.kill();
```

worker.id

Added in: v0.8.0

- <number>

Each new worker is given its own unique id, this id is stored in the `id`.

While a worker is alive, this is the key that indexes it in cluster.workers

worker.isConnected()

Added in: v0.11.14

This function returns `true` if the worker is connected to its master via its IPC channel, `false` otherwise. A worker is connected to its master after it has been created. It is disconnected after the `'disconnect'` event is emitted.

worker.isDead()

Added in: v0.11.14

This function returns `true` if the worker's process has terminated (either because of exiting or being signaled). Otherwise, it returns `false`.

worker.kill([signal='SIGTERM'])

Added in: v0.9.12

- `signal` <string> Name of the kill signal to send to the worker process.

This function will kill the worker. In the master, it does this by disconnecting the `worker.process`, and once disconnected, killing with `signal`. In the worker, it does it by disconnecting the channel, and then exiting with code `0`.

Causes `.exitedAfterDisconnect` to be set.

This method is aliased as `worker.destroy()` for backwards compatibility.

Note that in a worker, `process.kill()` exists, but it is not this function, it is kill.

worker.process

Added in: v0.7.0

- <ChildProcess>

All workers are created using child_process.fork(), the returned object from this function is stored as `.process`. In a worker, the global `process` is stored.

See: Child Process module

Note that workers will call `process.exit(0)` if the `'disconnect'` event occurs on `process` and `.exitedAfterDisconnect` is not `true`. This protects against accidental disconnection.

worker.send(message[, sendHandle][, callback])

▶ History

- `message` <Object>
- `sendHandle` <Handle>
- `callback` <Function>
- Returns: <boolean>

Send a message to a worker or master, optionally with a handle.

In the master this sends a message to a specific worker. It is identical to ChildProcess.send() .

In a worker this sends a message to the master. It is identical to `process.send()` .

This example will echo back all messages from the master:

```
if (cluster.isMaster) {
  const worker = cluster.fork();
  worker.send('hi there');

} else if (cluster.isWorker) {
  process.on('message', (msg) => {
    process.send(msg);
  });
}
```

Event: 'disconnect'

Added in: v0.7.9

- `worker` <cluster.Worker>

Emitted after the worker IPC channel has disconnected. This can occur when a worker exits gracefully, is killed, or is disconnected manually (such as with worker.disconnect()).

There may be a delay between the `'disconnect'` and `'exit'` events. These events can be used to detect if the process is stuck in a cleanup or if there are long-living connections.

```
cluster.on('disconnect', (worker) => {
  console.log(`The worker #${worker.id} has disconnected`);
});
```

Event: 'exit'

Added in: v0.7.9

- `worker` <cluster.Worker>
- `code` <number> The exit code, if it exited normally.
- `signal` <string> The name of the signal (e.g. `'SIGHUP'`) that caused the process to be killed.

When any of the workers die the cluster module will emit the `'exit'` event.

This can be used to restart the worker by calling `.fork()` again.

```
cluster.on('exit', (worker, code, signal) => {
  console.log('worker %d died (%s). restarting...',
              worker.process.pid, signal || code);
  cluster.fork();
});
```

See child_process event: 'exit' .

Event: 'fork'

Added in: v0.7.0

- `worker` <cluster.Worker>

When a new worker is forked the cluster module will emit a `'fork'` event. This can be used to log worker activity, and create a custom timeout.

```
const timeouts = [];
function errorMsg() {
  console.error('Something must be wrong with the connection ...');
}

cluster.on('fork', (worker) => {
  timeouts[worker.id] = setTimeout(errorMsg, 2000);
});
cluster.on('listening', (worker, address) => {
  clearTimeout(timeouts[worker.id]);
});
cluster.on('exit', (worker, code, signal) => {
  clearTimeout(timeouts[worker.id]);
  errorMsg();
});
```

Event: 'listening'

Added in: v0.7.0

- `worker` <cluster.Worker>
- `address` <Object>

After calling `listen()` from a worker, when the `'listening'` event is emitted on the server a `'listening'` event will also be emitted on `cluster` in the master.

The event handler is executed with two arguments, the `worker` contains the worker object and the `address` object contains the following connection properties: `address`, `port` and `addressType`. This is very useful if the worker is listening on more than one address.

```
cluster.on('listening', (worker, address) => {
  console.log(
    `A worker is now connected to ${address.address}:${address.port}`);
});
```

The `addressType` is one of:

- `4` (TCPv4)
- `6` (TCPv6)
- `-1` (unix domain socket)
- `'udp4'` or `'udp6'` (UDP v4 or v6)

Event: 'message'

▸ History

- `worker` <cluster.Worker>
- `message` <Object>
- `handle` <undefined> | <Object>

Emitted when the cluster master receives a message from any worker.

See child_process event: 'message'.

Before Node.js v6.0, this event emitted only the message and the handle, but not the worker object, contrary to what the documentation stated.

If support for older versions is required but a worker object is not required, it is possible to work around the discrepancy by checking the number of arguments:

```
cluster.on('message', (worker, message, handle) => {
  if (arguments.length === 2) {
    handle = message;
    message = worker;
    worker = undefined;
  }
  // ...
});
```

Event: 'online'

Added in: v0.7.0

- `worker` <cluster.Worker>

After forking a new worker, the worker should respond with an online message. When the master receives an online message it will emit this event. The difference between `'fork'` and `'online'` is that fork is emitted when the master forks a worker, and 'online' is emitted when the worker is running.

```
cluster.on('online', (worker) => {
  console.log('Yay, the worker responded after it was forked');
});
```

Event: 'setup'

Added in: v0.7.1

- `settings` <Object>

Emitted every time `.setupMaster()` is called.

The `settings` object is the `cluster.settings` object at the time `.setupMaster()` was called and is advisory only, since multiple calls to `.setupMaster()` can be made in a single tick.

If accuracy is important, use `cluster.settings`.

cluster.disconnect([callback])

Added in: v0.7.7

- `callback` <Function> Called when all workers are disconnected and handles are closed.

Calls `.disconnect()` on each worker in `cluster.workers`.

When they are disconnected all internal handles will be closed, allowing the master process to die gracefully if no other event is waiting.

The method takes an optional callback argument which will be called when finished.

This can only be called from the master process.

cluster.fork([env])

Added in: v0.6.0

- `env` <Object> Key/value pairs to add to worker process environment.
- Returns: <cluster.Worker>

Spawn a new worker process.

This can only be called from the master process.

cluster.isMaster

Added in: v0.8.1

- <boolean>

True if the process is a master. This is determined by the `process.env.NODE_UNIQUE_ID`. If `process.env.NODE_UNIQUE_ID` is undefined, then `isMaster` is `true`.

cluster.isWorker

Added in: v0.6.0

- <boolean>

True if the process is not a master (it is the negation of `cluster.isMaster`).

cluster.schedulingPolicy

Added in: v0.11.2

The scheduling policy, either `cluster.SCHED_RR` for round-robin or `cluster.SCHED_NONE` to leave it to the operating system. This is a global setting and effectively frozen once either the first worker is spawned, or `cluster.setupMaster()` is called, whichever comes first.

`SCHED_RR` is the default on all operating systems except Windows. Windows will change to `SCHED_RR` once libuv is able to effectively distribute IOCP handles without incurring a large performance hit.

`cluster.schedulingPolicy` can also be set through the `NODE_CLUSTER_SCHED_POLICY` environment variable. Valid values are `'rr'` and `'none'`.

cluster.settings

▸ History

- <Object>
  - `execArgv` <Array> List of string arguments passed to the Node.js executable. **Default:** `process.execArgv`
  - `exec` <string> File path to worker file. **Default:** `process.argv[1]`
  - `args` <Array> String arguments passed to worker. **Default:** `process.argv.slice(2)`
  - `cwd` <string> Current working directory of the worker process. **Default:** `undefined` (inherits from parent process)
  - `silent` <boolean> Whether or not to send output to parent's stdio. **Default:** `false`
  - `stdio` <Array> Configures the stdio of forked processes. Because the cluster module relies on IPC to function, this configuration must contain an `'ipc'` entry. When this option is provided, it overrides `silent`.
  - `uid` <number> Sets the user identity of the process. (See setuid(2).)
  - `gid` <number> Sets the group identity of the process. (See setgid(2).)
  - `inspectPort` <number> | <Function> Sets inspector port of worker. This can be a number, or a function that takes no arguments and

returns a number. By default each worker gets its own port, incremented from the master's `process.debugPort`.

  - `windowsHide` <boolean> Hide the forked processes console window that would normally be created on Windows systems. **Default:** `false`

After calling `.setupMaster()` (or `.fork()`) this settings object will contain the settings, including the default values.

This object is not intended to be changed or set manually.

cluster.setupMaster([settings])

▸ History

- `settings` <Object> see cluster.settings

`setupMaster` is used to change the default 'fork' behavior. Once called, the settings will be present in `cluster.settings`.

Note that:

- Any settings changes only affect future calls to `.fork()` and have no effect on workers that are already running.
- The *only* attribute of a worker that cannot be set via `.setupMaster()` is the `env` passed to `.fork()`.
- The defaults above apply to the first call only, the defaults for later calls is the current value at the time of `cluster.setupMaster()` is called.

Example:

```
const cluster = require('cluster');
cluster.setupMaster({
  exec: 'worker.js',
  args: ['--use', 'https'],
  silent: true
});
cluster.fork(); // https worker
cluster.setupMaster({
  exec: 'worker.js',
  args: ['--use', 'http']
});
cluster.fork(); // http worker
```

This can only be called from the master process.

cluster.worker

Added in: v0.7.0

- <Object>

A reference to the current worker object. Not available in the master process.

```
const cluster = require('cluster');

if (cluster.isMaster) {
  console.log('I am master');
  cluster.fork();
  cluster.fork();
} else if (cluster.isWorker) {
  console.log(`I am worker #${cluster.worker.id}`);
}
```

cluster.workers

Added in: v0.7.0

- <Object>

A hash that stores the active worker objects, keyed by `id` field. Makes it easy to loop through all the workers. It is only available in the master process.

A worker is removed from cluster.workers after the worker has disconnected *and* exited. The order between these two events cannot be determined in advance. However, it is guaranteed that the removal from the cluster.workers list happens before last `'disconnect'` or `'exit'` event is emitted.

```
// Go through all workers
function eachWorker(callback) {
  for (const id in cluster.workers) {
    callback(cluster.workers[id]);
  }
}
eachWorker((worker) => {
  worker.send('big announcement to all workers');
});
```

Using the worker's unique id is the easiest way to locate the worker.

```
socket.on('data', (id) => {
  const worker = cluster.workers[id];
});
```

Command Line Options

Node.js comes with a variety of CLI options. These options expose built-in debugging, multiple ways to execute scripts, and other helpful runtime options.

To view this documentation as a manual page in a terminal, run `man node`.

Synopsis

`node [options] [V8 options] [script.js | -e "script" | -] [--] [arguments]`

`node debug [script.js | -e "script" | <host>:<port>] …`

`node --v8-options`

Execute without arguments to start the REPL.

For more info about `node debug`*, please see the* *debugger* *documentation.*

Options

`-v`, `--version`

Added in: v0.1.3

Print node's version.

`-h`, `--help`

Added in: v0.1.3

Print node command line options. The output of this option is less detailed than this document.

`-e`, `--eval "script"`

▸ History

Evaluate the following argument as JavaScript. The modules which are predefined in the REPL can also be used in `script`.

On Windows, using `cmd.exe` a single quote will not work correctly because it only recognizes double `"` for quoting. In Powershell or Git bash, both `'` and `"` are usable.

`-p`, `--print "script"`

▸ History

Identical to `-e` but prints the result.

`-c`, `--check`

Added in: v5.0.0, v4.2.0

Syntax check the script without executing.

`-i`, `--interactive`

Added in: v0.7.7

Opens the REPL even if stdin does not appear to be a terminal.

`-r`, `--require module`

Added in: v1.6.0

Preload the specified module at startup.

Follows `require()`'s module resolution rules. `module` may be either a path to a file, or a node module name.

`--inspect[=[host:]port]`

Added in: v6.3.0

Activate inspector on host:port. Default is 127.0.0.1:9229.

V8 inspector integration allows tools such as Chrome DevTools and IDEs to debug and profile Node.js instances. The tools attach to Node.js instances via a tcp port and communicate using the Chrome Debugging Protocol .

`--inspect-brk[=[host:]port]`

Added in: v7.6.0

Activate inspector on host:port and break at start of user script. Default host:port is 127.0.0.1:9229.

`--inspect-port=[host:]port`

Added in: v7.6.0

Set the host:port to be used when the inspector is activated. Useful when activating the inspector by sending the `SIGUSR1` signal.

Default host is 127.0.0.1.

`--no-deprecation`

Added in: v0.8.0

Silence deprecation warnings.

`--trace-deprecation`

Added in: v0.8.0

Print stack traces for deprecations.

`--throw-deprecation`

Added in: v0.11.14

Throw errors for deprecations.

`--pending-deprecation`

Added in: v8.0.0

Emit pending deprecation warnings.

Pending deprecations are generally identical to a runtime deprecation with the notable exception that they are turned*off* by default and will not be emitted unless either the `--pending-deprecation` command line flag, or the `NODE_PENDING_DEPRECATION=1` environment variable, is set. Pending deprecations are used to provide a kind of selective "early warning" mechanism that developers may leverage to detect deprecated API usage.

`--no-warnings`

Added in: v6.0.0

Silence all process warnings (including deprecations).

`--abort-on-uncaught-exception`

Added in: v0.10

Aborting instead of exiting causes a core file to be generated for post-mortem analysis using a debugger (such as `lldb`, `gdb`, and `mdb`).

If this flag is passed, the behavior can still be set to not abort through process.setUncaughtExceptionCaptureCallback() (and through usage of the `domain` module that uses it).

`--trace-warnings`

Added in: v6.0.0

Print stack traces for process warnings (including deprecations).

`--redirect-warnings=file`

Added in: v8.0.0

Write process warnings to the given file instead of printing to stderr. The file will be created if it does not exist, and will be appended to if it does. If an error occurs while attempting to write the warning to the file, the warning will be written to stderr instead.

`--trace-sync-io`

Added in: v2.1.0

Prints a stack trace whenever synchronous I/O is detected after the first turn of the event loop.

`--no-force-async-hooks-checks`

Added in: v9.0.0

Disables runtime checks for `async_hooks`. These will still be enabled dynamically when `async_hooks` is enabled.

`--trace-events-enabled`

Added in: v7.7.0

Enables the collection of trace event tracing information.

--trace-event-categories

Added in: v7.7.0

A comma separated list of categories that should be traced when trace event tracing is enabled using `--trace-events-enabled`.

--trace-event-file-pattern

Added in: v9.8.0

Template string specifying the filepath for the trace event data, it supports `${rotation}` and `${pid}`.

--zero-fill-buffers

Added in: v6.0.0

Automatically zero-fills all newly allocated Buffer and SlowBuffer instances.

--preserve-symlinks

Added in: v6.3.0

Instructs the module loader to preserve symbolic links when resolving and caching modules.

By default, when Node.js loads a module from a path that is symbolically linked to a different on-disk location, Node.js will dereference the link and use the actual on-disk "real path" of the module as both an identifier and as a root path to locate other dependency modules. In most cases, this default behavior is acceptable. However, when using symbolically linked peer dependencies, as illustrated in the example below, the default behavior causes an exception to be thrown if `moduleA` attempts to require `moduleB` as a peer dependency:

```
{appDir}
 ├── app
 │   ├── index.js
 │   └── node_modules
 │       ├── moduleA -> {appDir}/moduleA
 │       └── moduleB
 │           ├── index.js
 │           └── package.json
 └── moduleA
     ├── index.js
     └── package.json
```

The `--preserve-symlinks` command line flag instructs Node.js to use the symlink path for modules as opposed to the real path, allowing symbolically linked peer dependencies to be found.

Note, however, that using `--preserve-symlinks` can have other side effects. Specifically, symbolically linked *native* modules can fail to load if those are linked from more than one location in the dependency tree (Node.js would see those as two separate modules and would attempt to load the module multiple times, causing an exception to be thrown).

--track-heap-objects

Added in: v2.4.0

Track heap object allocations for heap snapshots.

--prof-process

Added in: v5.2.0

Process V8 profiler output generated using the V8 option `--prof`.

`--v8-options`

Added in: v0.1.3

Print V8 command line options.

V8 options allow words to be separated by both dashes (`-`) or underscores (`_`).

For example, `--stack-trace-limit` is equivalent to `--stack_trace_limit`.

`--tls-cipher-list=list`

Added in: v4.0.0

Specify an alternative default TLS cipher list. (Requires Node.js to be built with crypto support. (Default))

`--enable-fips`

Added in: v6.0.0

Enable FIPS-compliant crypto at startup. (Requires Node.js to be built with `./configure --openssl-fips`)

`--force-fips`

Added in: v6.0.0

Force FIPS-compliant crypto on startup. (Cannot be disabled from script code.) (Same requirements as `--enable-fips`)

`--openssl-config=file`

Added in: v6.9.0

Load an OpenSSL configuration file on startup. Among other uses, this can be used to enable FIPS-compliant crypto if Node.js is built with `./configure --openssl-fips`.

`--use-openssl-ca`, `--use-bundled-ca`

Added in: v6.11.0

Use OpenSSL's default CA store or use bundled Mozilla CA store as supplied by current Node.js version. The default store is selectable at build-time.

Using OpenSSL store allows for external modifications of the store. For most Linux and BSD distributions, this store is maintained by the distribution maintainers and system administrators. OpenSSL CA store location is dependent on configuration of the OpenSSL library but this can be altered at runtime using environment variables.

The bundled CA store, as supplied by Node.js, is a snapshot of Mozilla CA store that is fixed at release time. It is identical on all supported platforms.

See `SSL_CERT_DIR` and `SSL_CERT_FILE`.

`--icu-data-dir=file`

Added in: v0.11.15

Specify ICU data load path. (overrides `NODE_ICU_DATA`)

`-`

Added in: v8.0.0

Alias for stdin, analogous to the use of - in other command line utilities, meaning that the script will be read from stdin, and the rest of the options are passed to that script.

`--`

Added in: v6.11.0

Indicate the end of node options. Pass the rest of the arguments to the script. If no script filename or eval/print script is supplied prior to this, then the next argument will be used as a script filename.

Environment Variables

`NODE_DEBUG=module[,…]`

Added in: v0.1.32

`','`-separated list of core modules that should print debug information.

`NODE_PATH=path[:…]`

Added in: v0.1.32

`':'`-separated list of directories prefixed to the module search path.

On Windows, this is a `';'`-separated list instead.

`NODE_DISABLE_COLORS=1`

Added in: v0.3.0

When set to `1` colors will not be used in the REPL.

`NODE_ICU_DATA=file`

Added in: v0.11.15

Data path for ICU (Intl object) data. Will extend linked-in data when compiled with small-icu support.

`NODE_NO_WARNINGS=1`

Added in: v6.11.0

When set to `1`, process warnings are silenced.

`NODE_OPTIONS=options...`

Added in: v8.0.0

A space-separated list of command line options. `options...` are interpreted as if they had been specified on the command line before the actual command line (so they can be overridden). Node will exit with an error if an option that is not allowed in the environment is used, such as `-p` or a script file.

Node options that are allowed are:

- `--enable-fips`
- `--force-fips`
- `--icu-data-dir`
- `--inspect-brk`
- `--inspect-port`
- `--inspect`
- `--no-deprecation`
- `--no-warnings`
- `--openssl-config`
- `--redirect-warnings`
- `--require`, `-r`

- `--throw-deprecation`
- `--tls-cipher-list`
- `--trace-deprecation`
- `--trace-events-categories`
- `--trace-events-enabled`
- `--trace-event-file-pattern`
- `--trace-sync-io`
- `--trace-warnings`
- `--track-heap-objects`
- `--use-bundled-ca`
- `--use-openssl-ca`
- `--v8-pool-size`
- `--zero-fill-buffers`

V8 options that are allowed are:

- `--abort-on-uncaught-exception`
- `--max-old-space-size`
- `--perf-basic-prof`
- `--perf-prof`
- `--stack-trace-limit`

`NODE_PENDING_DEPRECATION=1`

Added in: v8.0.0

When set to `1`, emit pending deprecation warnings.

Pending deprecations are generally identical to a runtime deprecation with the notable exception that they are turned*off* by default and will not be emitted unless either the `--pending-deprecation` command line flag, or the `NODE_PENDING_DEPRECATION=1` environment variable, is set. Pending deprecations are used to provide a kind of selective "early warning" mechanism that developers may leverage to detect deprecated API usage.

`NODE_PRESERVE_SYMLINKS=1`

Added in: v7.1.0

When set to `1`, instructs the module loader to preserve symbolic links when resolving and caching modules.

`NODE_REPL_HISTORY=file`

Added in: v3.0.0

Path to the file used to store the persistent REPL history. The default path is `~/.node_repl_history`, which is overridden by this variable. Setting the value to an empty string (`''` or `' '`) disables persistent REPL history.

`NODE_EXTRA_CA_CERTS=file`

Added in: v7.3.0

When set, the well known "root" CAs (like VeriSign) will be extended with the extra certificates in `file`. The file should consist of one or more trusted certificates in PEM format. A message will be emitted (once) with `process.emitWarning()` if the file is missing or malformed, but any errors are otherwise ignored.

Note that neither the well known nor extra certificates are used when the `ca` options property is explicitly specified for a TLS or HTTPS client or server.

`OPENSSL_CONF=file`

Added in: v6.11.0

Load an OpenSSL configuration file on startup. Among other uses, this can be used to enable FIPS-compliant crypto if Node.js is built with `./configure --openssl-fips`.

If the `--openssl-config` command line option is used, the environment variable is ignored.

`SSL_CERT_DIR=dir`

Added in: v7.7.0

If `--use-openssl-ca` is enabled, this overrides and sets OpenSSL's directory containing trusted certificates.

Be aware that unless the child environment is explicitly set, this environment variable will be inherited by any child processes, and if they use OpenSSL, it may cause them to trust the same CAs as node.

`SSL_CERT_FILE=file`

Added in: v7.7.0

If `--use-openssl-ca` is enabled, this overrides and sets OpenSSL's file containing trusted certificates.

Be aware that unless the child environment is explicitly set, this environment variable will be inherited by any child processes, and if they use OpenSSL, it may cause them to trust the same CAs as node.

`NODE_REDIRECT_WARNINGS=file`

Added in: v8.0.0

When set, process warnings will be emitted to the given file instead of printing to stderr. The file will be created if it does not exist, and will be appended to if it does. If an error occurs while attempting to write the warning to the file, the warning will be written to stderr instead. This is equivalent to using the `--redirect-warnings=file` command-line flag.

`UV_THREADPOOL_SIZE=size`

Set the number of threads used in libuv's threadpool to `size` threads.

Asynchronous system APIs are used by Node.js whenever possible, but where they do not exist, libuv's threadpool is used to create asynchronous node APIs based on synchronous system APIs. Node.js APIs that use the threadpool are:

- all `fs` APIs, other than the file watcher APIs and those that are explicitly synchronous
- `crypto.pbkdf2()`
- `crypto.randomBytes()`, unless it is used without a callback
- `crypto.randomFill()`
- `dns.lookup()`
- all `zlib` APIs, other than those that are explicitly synchronous

Because libuv's threadpool has a fixed size, it means that if for whatever reason any of these APIs takes a long time, other (seemingly unrelated) APIs that run in libuv's threadpool will experience degraded performance. In order to mitigate this issue, one potential solution is to increase the size of libuv's threadpool by setting the `'UV_THREADPOOL_SIZE'` environment variable to a value greater than `4` (its current default value). For more information, see the libuv threadpool documentation.

Console

Stability: 2 - Stable

The `console` module provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers.

The module exports two specific components:

- A `Console` class with methods such as `console.log()`, `console.error()` and `console.warn()` that can be used to write to any Node.js stream.
- A global `console` instance configured to write to process.stdout and process.stderr. The global `console` can be used without calling `require('console')`.

Warning: The global console object's methods are neither consistently synchronous like the browser APIs they resemble, nor are they consistently asynchronous like all other Node.js streams. See the note on process I/O for more information.

Example using the global `console`:

```
console.log('hello world');
// Prints: hello world, to stdout
console.log('hello %s', 'world');
// Prints: hello world, to stdout
console.error(new Error('Whoops, something bad happened'));
// Prints: [Error: Whoops, something bad happened], to stderr

const name = 'Will Robinson';
console.warn(`Danger ${name}! Danger!`);
// Prints: Danger Will Robinson! Danger!, to stderr
```

Example using the `Console` class:

```
const out = getStreamSomehow();
const err = getStreamSomehow();
const myConsole = new console.Console(out, err);

myConsole.log('hello world');
// Prints: hello world, to out
myConsole.log('hello %s', 'world');
// Prints: hello world, to out
myConsole.error(new Error('Whoops, something bad happened'));
// Prints: [Error: Whoops, something bad happened], to err

const name = 'Will Robinson';
myConsole.warn(`Danger ${name}! Danger!`);
// Prints: Danger Will Robinson! Danger!, to err
```

Class: Console

▸ History

The `Console` class can be used to create a simple logger with configurable output streams and can be accessed using either `require('console').Console` or `console.Console` (or their destructured counterparts):

```
const { Console } = require('console');
```

```
const { Console } = console;
```

new Console(stdout[, stderr])

- `stdout` <stream.Writable>
- `stderr` <stream.Writable>

Creates a new `Console` with one or two writable stream instances. `stdout` is a writable stream to print log or info output. `stderr` is used for warning or error output. If `stderr` is not provided, `stdout` is used for `stderr`.

```
const output = fs.createWriteStream('./stdout.log');
const errorOutput = fs.createWriteStream('./stderr.log');
// custom simple logger
const logger = new Console(output, errorOutput);
// use it like console
const count = 5;
logger.log('count: %d', count);
// in stdout.log: count 5
```

The global `console` is a special `Console` whose output is sent to process.stdout and process.stderr. It is equivalent to calling:

```
new Console(process.stdout, process.stderr);
```

console.assert(value[, message][, ...args])

Added in: v0.1.101

- `value` <any>
- `message` <any>
- `...args` <any>

A simple assertion test that verifies whether `value` is truthy. If it is not, an `AssertionError` is thrown. If provided, the error `message` is formatted using util.format() and used as the error message.

```
console.assert(true, 'does nothing');
// OK
console.assert(false, 'Whoops %s', 'didn\'t work');
// AssertionError: Whoops didn't work
```

The `console.assert()` method is implemented differently in Node.js than the `console.assert()` method available in browsers.

Specifically, in browsers, calling `console.assert()` with a falsy assertion will cause the `message` to be printed to the console without interrupting execution of subsequent code. In Node.js, however, a falsy assertion will cause an `AssertionError` to be thrown.

Functionality approximating that implemented by browsers can be implemented by extending Node.js' `console` and overriding the `console.assert()` method.

In the following example, a simple module is created that extends and overrides the default behavior of `console` in Node.js.

```js
'use strict';

// Creates a simple extension of console with a
// new impl for assert without monkey-patching.
const myConsole = Object.create(console, {
  assert: {
    value: function assert(assertion, message, ...args) {
      try {
        console.assert(assertion, message, ...args);
      } catch (err) {
        console.error(err.stack);
      }
    },
    configurable: true,
    enumerable: true,
    writable: true,
  },
});

module.exports = myConsole;
```

This can then be used as a direct replacement for the built in console:

```js
const console = require('./myConsole');
console.assert(false, 'this message will print, but no error thrown');
console.log('this will also print');
```

console.clear()

Added in: v8.3.0

When `stdout` is a TTY, calling `console.clear()` will attempt to clear the TTY. When `stdout` is not a TTY, this method does nothing.

The specific operation of `console.clear()` can vary across operating systems and terminal types. For most Linux operating systems, `console.clear()` operates similarly to the `clear` shell command. On Windows, `console.clear()` will clear only the output in the current terminal viewport for the Node.js binary.

console.count([label])

Added in: v8.3.0

- `label` <string> The display label for the counter. Defaults to `'default'`.

Maintains an internal counter specific to `label` and outputs to `stdout` the number of times `console.count()` has been called with the given `label`.

```
> console.count()
default: 1
undefined
> console.count('default')
default: 2
undefined
> console.count('abc')
abc: 1
undefined
> console.count('xyz')
xyz: 1
undefined
> console.count('abc')
abc: 2
undefined
> console.count()
default: 3
undefined
>
```

console.countReset([label='default'])

Added in: v8.3.0

- `label` <string> The display label for the counter. Defaults to `'default'`.

Resets the internal counter specific to `label`.

```
> console.count('abc');
abc: 1
undefined
> console.countReset('abc');
undefined
> console.count('abc');
abc: 1
undefined
>
```

console.debug(data[, ...args])

▶ History

- `data` <any>
- `...args` <any>

The `console.debug()` function is an alias for `console.log()`.

console.dir(obj[, options])

Added in: v0.1.101

- `obj` <any>
- `options` <Object>
  - `showHidden` <boolean>
  - `depth` <number>
  - `colors` <boolean>

Uses util.inspect() on obj and prints the resulting string to stdout. This function bypasses any custom inspect() function defined on obj. An optional options object may be passed to alter certain aspects of the formatted string:

- showHidden - if true then the object's non-enumerable and symbol properties will be shown too. Defaults to false.
- depth - tells util.inspect() how many times to recurse while formatting the object. This is useful for inspecting large complicated objects. Defaults to 2. To make it recurse indefinitely, pass null.
- colors - if true, then the output will be styled with ANSI color codes. Defaults to false. Colors are customizable; see customizing util.inspect() colors.

console.dirxml(...data)

▸ History

- ...data <any>

This method calls console.log() passing it the arguments received. Please note that this method does not produce any XML formatting.

console.error([data][, ...args])

Added in: v0.1.100

- data <any>
- ...args <any>

Prints to stderr with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to printf(3) (the arguments are all passed to util.format()).

```
const code = 5;
console.error('error #%d', code);
// Prints: error #5, to stderr
console.error('error', code);
// Prints: error 5, to stderr
```

If formatting elements (e.g. %d) are not found in the first string then util.inspect() is called on each argument and the resulting string values are concatenated. See util.format() for more information.

console.group([...label])

Added in: v8.5.0

- ...label <any>

Increases indentation of subsequent lines by two spaces.

If one or more label s are provided, those are printed first without the additional indentation.

console.groupCollapsed()

Added in: v8.5.0

An alias for console.group().

console.groupEnd()

Added in: v8.5.0

Decreases indentation of subsequent lines by two spaces.

console.info([data][, ...args])

Added in: v0.1.100

- data <any>
- ...args <any>

The console.info() function is an alias for console.log() .

console.log([data][, ...args])

Added in: v0.1.100

- data <any>
- ...args <any>

Prints to stdout with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to printf(3) (the arguments are all passed to util.format()).

```
const count = 5;
console.log('count: %d', count);
// Prints: count: 5, to stdout
console.log('count:', count);
// Prints: count: 5, to stdout
```

See util.format() for more information.

console.time(label)

Added in: v0.1.104

- label <string> Defaults to 'default'.

Starts a timer that can be used to compute the duration of an operation. Timers are identified by a unique label . Use the same label when calling console.timeEnd() to stop the timer and output the elapsed time in milliseconds to stdout . Timer durations are accurate to the sub-millisecond.

console.timeEnd(label)

▸ History

- label <string> Defaults to 'default'.

Stops a timer that was previously started by calling console.time() and prints the result to stdout :

```
console.time('100-elements');
for (let i = 0; i < 100; i++) {}
console.timeEnd('100-elements');
// prints 100-elements: 225.438ms
```

console.trace([message][, ...args])

Added in: v0.1.104

- message <any>
- ...args <any>

Prints to stderr the string 'Trace :' , followed by the util.format() formatted message and stack trace to the current position in the code.

```
console.trace('Show me');
// Prints: (stack trace will vary based on where trace is called)
//  Trace: Show me
//    at repl:2:9
//    at REPLServer.defaultEval (repl.js:248:27)
//    at bound (domain.js:287:14)
//    at REPLServer.runBound [as eval] (domain.js:300:12)
//    at REPLServer.<anonymous> (repl.js:412:12)
//    at emitOne (events.js:82:20)
//    at REPLServer.emit (events.js:169:7)
//    at REPLServer.Interface._onLine (readline.js:210:10)
//    at REPLServer.Interface._line (readline.js:549:8)
//    at REPLServer.Interface._ttyWrite (readline.js:826:14)
```

console.warn([data][, ...args])

Added in: v0.1.100

- `data` <any>
- `...args` <any>

The `console.warn()` function is an alias for `console.error()`.

Inspector only methods

The following methods are exposed by the V8 engine in the general API but do not display anything unless used in conjunction with the inspector (`--inspect` flag).

console.markTimeline(label)

Added in: v8.0.0

- `label` <string> Defaults to `'default'`.

This method does not display anything unless used in the inspector. The `console.markTimeline()` method is the deprecated form of `console.timeStamp()`.

console.profile([label])

Added in: v8.0.0

- `label` <string>

This method does not display anything unless used in the inspector. The `console.profile()` method starts a JavaScript CPU profile with an optional label until `console.profileEnd()` is called. The profile is then added to the **Profile** panel of the inspector.

```
console.profile('MyLabel');
// Some code
console.profileEnd();
// Adds the profile 'MyLabel' to the Profiles panel of the inspector.
```

console.profileEnd()

Added in: v8.0.0

This method does not display anything unless used in the inspector. Stops the current JavaScript CPU profiling session if one has been started and prints the report to the **Profiles** panel of the inspector. See `console.profile()` for an example.

console.table(array[, columns])

Added in: v8.0.0

- `array` <Array> | <Object>
- `columns` <Array>

This method does not display anything unless used in the inspector. Prints to `stdout` the array `array` formatted as a table.

console.timeStamp([label])

Added in: v8.0.0

- `label` <string>

This method does not display anything unless used in the inspector. The `console.timeStamp()` method adds an event with the label `label` to the **Timeline** panel of the inspector.

console.timeline([label])

Added in: v8.0.0

- `label` <string> Defaults to `'default'`.

This method does not display anything unless used in the inspector. The `console.timeline()` method is the deprecated form of console.time() .

console.timelineEnd([label])

Added in: v8.0.0

- `label` <string> Defaults to `'default'`.

This method does not display anything unless used in the inspector. The `console.timelineEnd()` method is the deprecated form of console.timeEnd() .

Crypto

Stability: 2 - Stable

The `crypto` module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign, and verify functions.

Use `require('crypto')` to access this module.

```
const crypto = require('crypto');

const secret = 'abcdefg';
const hash = crypto.createHmac('sha256', secret)
                   .update('I love cupcakes')
                   .digest('hex');
console.log(hash);
// Prints:
//   c0fa1bc00531bd78ef38c628449c5102aeabd49b5dc3a2a516ea6ea959d6658e
```

Determining if crypto support is unavailable

It is possible for Node.js to be built without including support for the `crypto` module. In such cases, calling `require('crypto')` will result in an error being thrown.

```
let crypto;
try {
  crypto = require('crypto');
} catch (err) {
  console.log('crypto support is disabled!');
}
```

Class: Certificate

Added in: v0.11.8

SPKAC is a Certificate Signing Request mechanism originally implemented by Netscape and was specified formally as part of HTML5's keygen element.

Note that `<keygen>` is deprecated since HTML 5.2 and new projects should not use this element anymore.

The `crypto` module provides the `Certificate` class for working with SPKAC data. The most common usage is handling output generated by the HTML5 `<keygen>` element. Node.js uses OpenSSL's SPKAC implementation internally.

Certificate.exportChallenge(spkac)

Added in: v9.0.0

- `spkac` <string> | <Buffer> | <TypedArray> | <DataView>
- Returns <Buffer> The challenge component of the `spkac` data structure, which includes a public key and a challenge.

```
const { Certificate } = require('crypto');
const spkac = getSpkacSomehow();
const challenge = Certificate.exportChallenge(spkac);
console.log(challenge.toString('utf8'));
// Prints: the challenge as a UTF8 string
```

Certificate.exportPublicKey(spkac[, encoding])

Added in: v9.0.0

- `spkac` <string> | <Buffer> | <TypedArray> | <DataView>
- `encoding` <string>
- Returns <Buffer> The public key component of the `spkac` data structure, which includes a public key and a challenge.

```
const { Certificate } = require('crypto');
const spkac = getSpkacSomehow();
const publicKey = Certificate.exportPublicKey(spkac);
console.log(publicKey);
// Prints: the public key as <Buffer ...>
```

Certificate.verifySpkac(spkac)

Added in: v9.0.0

- `spkac` <Buffer> | <TypedArray> | <DataView>
- Returns <boolean> `true` if the given `spkac` data structure is valid, `false` otherwise.

```
const { Certificate } = require('crypto');
const spkac = getSpkacSomehow();
console.log(Certificate.verifySpkac(Buffer.from(spkac)));
// Prints: true or false
```

Legacy API

As a still supported legacy interface, it is possible (but not recommended) to create new instances of the `crypto.Certificate` class as illustrated in the examples below.

new crypto.Certificate()

Instances of the `Certificate` class can be created using the `new` keyword or by calling `crypto.Certificate()` as a function:

```
const crypto = require('crypto');

const cert1 = new crypto.Certificate();
const cert2 = crypto.Certificate();
```

certificate.exportChallenge(spkac)

Added in: v0.11.8

- `spkac` <string> | <Buffer> | <TypedArray> | <DataView>
- Returns <Buffer> The challenge component of the `spkac` data structure, which includes a public key and a challenge.

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const challenge = cert.exportChallenge(spkac);
console.log(challenge.toString('utf8'));
// Prints: the challenge as a UTF8 string
```

certificate.exportPublicKey(spkac)

Added in: v0.11.8

- `spkac` <string> | <Buffer> | <TypedArray> | <DataView>
- Returns <Buffer> The public key component of the `spkac` data structure, which includes a public key and a challenge.

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const publicKey = cert.exportPublicKey(spkac);
console.log(publicKey);
// Prints: the public key as <Buffer ...>
```

certificate.verifySpkac(spkac)

Added in: v0.11.8

- `spkac` <Buffer> | <TypedArray> | <DataView>
- Returns <boolean> `true` if the given `spkac` data structure is valid, `false` otherwise.

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
console.log(cert.verifySpkac(Buffer.from(spkac)));
// Prints: true or false
```

Class: Cipher

Added in: v0.1.94

Instances of the `Cipher` class are used to encrypt data. The class can be used in one of two ways:

- As a stream that is both readable and writable, where plain unencrypted data is written to produce encrypted data on the readable side, or
- Using the `cipher.update()` and `cipher.final()` methods to produce the encrypted data.

The `crypto.createCipher()` or `crypto.createCipheriv()` methods are used to create `Cipher` instances. `Cipher` objects are not to be created directly using the `new` keyword.

Example: Using `Cipher` objects as streams:

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');

let encrypted = '';
cipher.on('readable', () => {
  const data = cipher.read();
  if (data)
    encrypted += data.toString('hex');
});
cipher.on('end', () => {
  console.log(encrypted);
  // Prints: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504
});

cipher.write('some clear text data');
cipher.end();
```

Example: Using `Cipher` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const cipher = crypto.createCipher('aes192', 'a password');

const input = fs.createReadStream('test.js');
const output = fs.createWriteStream('test.enc');

input.pipe(cipher).pipe(output);
```

Example: Using the `cipher.update()` and `cipher.final()` methods:

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');

let encrypted = cipher.update('some clear text data', 'utf8', 'hex');
encrypted += cipher.final('hex');
console.log(encrypted);
// Prints: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504
```

cipher.final([outputEncoding])

Added in: v0.1.94

- `outputEncoding` <string>

Returns any remaining enciphered contents. If `outputEncoding` parameter is one of `'latin1'`, `'base64'` or `'hex'`, a string is returned. If an `outputEncoding` is not provided, a `Buffer` is returned.

Once the `cipher.final()` method has been called, the `Cipher` object can no longer be used to encrypt data. Attempts to call `cipher.final()` more than once will result in an error being thrown.

cipher.setAAD(buffer)

Added in: v1.0.0

- `buffer` <Buffer>
- Returns the <Cipher> for method chaining.

When using an authenticated encryption mode (only `GCM` is currently supported), the `cipher.setAAD()` method sets the value used for the *additional authenticated data* (AAD) input parameter.

The `cipher.setAAD()` method must be called before `cipher.update()`.

cipher.getAuthTag()

Added in: v1.0.0

When using an authenticated encryption mode (only `GCM` is currently supported), the `cipher.getAuthTag()` method returns a `Buffer` containing the *authentication tag* that has been computed from the given data.

The `cipher.getAuthTag()` method should only be called after encryption has been completed using the `cipher.final()` method.

cipher.setAutoPadding([autoPadding])

Added in: v0.7.1

- `autoPadding` <boolean> Defaults to `true`.
- Returns the <Cipher> for method chaining.

When using block encryption algorithms, the `Cipher` class will automatically add padding to the input data to the appropriate block size. To disable the default padding call `cipher.setAutoPadding(false)`.

When `autoPadding` is `false`, the length of the entire input data must be a multiple of the cipher's block size or `cipher.final()` will throw an Error. Disabling automatic padding is useful for non-standard padding, for instance using `0x0` instead of PKCS padding.

The `cipher.setAutoPadding()` method must be called before `cipher.final()`.

cipher.update(data[, inputEncoding][, outputEncoding])

▸ History

- `data` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>

- `outputEncoding` <string>

Updates the cipher with `data`. If the `inputEncoding` argument is given, its value must be one of `'utf8'`, `'ascii'`, or `'latin1'` and the `data` argument is a string using the specified encoding. If the `inputEncoding` argument is not given, `data` must be a Buffer, `TypedArray`, or `DataView`. If `data` is a Buffer, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

The `outputEncoding` specifies the output format of the enciphered data, and can be `'latin1'`, `'base64'` or `'hex'`. If the `outputEncoding` is specified, a string using the specified encoding is returned. If no `outputEncoding` is provided, a Buffer is returned.

The `cipher.update()` method can be called multiple times with new data until cipher.final() is called. Calling `cipher.update()` after cipher.final() will result in an error being thrown.

Class: Decipher

Added in: v0.1.94

Instances of the `Decipher` class are used to decrypt data. The class can be used in one of two ways:

- As a stream that is both readable and writable, where plain encrypted data is written to produce unencrypted data on the readable side, or
- Using the decipher.update() and decipher.final() methods to produce the unencrypted data.

The crypto.createDecipher() or crypto.createDecipheriv() methods are used to create `Decipher` instances. `Decipher` objects are not to be created directly using the `new` keyword.

Example: Using `Decipher` objects as streams:

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');

let decrypted = '';
decipher.on('readable', () => {
  const data = decipher.read();
  if (data)
    decrypted += data.toString('utf8');
});
decipher.on('end', () => {
  console.log(decrypted);
  // Prints: some clear text data
});

const encrypted =
    'ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504';
decipher.write(encrypted, 'hex');
decipher.end();
```

Example: Using `Decipher` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const decipher = crypto.createDecipher('aes192', 'a password');

const input = fs.createReadStream('test.enc');
const output = fs.createWriteStream('test.js');

input.pipe(decipher).pipe(output);
```

Example: Using the decipher.update() and decipher.final() methods:

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');

const encrypted =
  'ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504';
let decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
// Prints: some clear text data
```

decipher.final([outputEncoding])

Added in: v0.1.94

- `outputEncoding` <string>

Returns any remaining deciphered contents. If `outputEncoding` parameter is one of `'latin1'`, `'ascii'` or `'utf8'`, a string is returned. If an `outputEncoding` is not provided, a `Buffer` is returned.

Once the `decipher.final()` method has been called, the `Decipher` object can no longer be used to decrypt data. Attempts to call `decipher.final()` more than once will result in an error being thrown.

decipher.setAAD(buffer)

▶ History

- `buffer` <Buffer> | <TypedArray> | <DataView>
- Returns the <Cipher> for method chaining.

When using an authenticated encryption mode (only `GCM` is currently supported), the `decipher.setAAD()` method sets the value used for the *additional authenticated data* (AAD) input parameter.

The `decipher.setAAD()` method must be called before `decipher.update()`.

decipher.setAuthTag(buffer)

▶ History

- `buffer` <Buffer> | <TypedArray> | <DataView>
- Returns the <Cipher> for method chaining.

When using an authenticated encryption mode (only `GCM` is currently supported), the `decipher.setAuthTag()` method is used to pass in the received *authentication tag*. If no tag is provided, or if the cipher text has been tampered with, `decipher.final()` will throw, indicating that the cipher text should be discarded due to failed authentication.

Note that this Node.js version does not verify the length of GCM authentication tags. Such a check *must* be implemented by applications and is crucial to the authenticity of the encrypted data, otherwise, an attacker can use an arbitrarily short authentication tag to increase the chances of successfully passing authentication (up to 0.39%). It is highly recommended to associate one of the values 16, 15, 14, 13, 12, 8 or 4 bytes with each key, and to only permit authentication tags of that length, see NIST SP 800-38D.

The `decipher.setAuthTag()` method must be called before `decipher.final()`.

decipher.setAutoPadding([autoPadding])

Added in: v0.7.1

- `autoPadding` <boolean> Defaults to `true`.
- Returns the <Cipher> for method chaining.

When data has been encrypted without standard block padding, calling `decipher.setAutoPadding(false)` will disable automatic padding to prevent `decipher.final()` from checking for and removing padding.

Turning auto padding off will only work if the input data's length is a multiple of the ciphers block size.

The `decipher.setAutoPadding()` method must be called before decipher.final() .

decipher.update(data[, inputEncoding][, outputEncoding])

▸ History

- `data` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>
- `outputEncoding` <string>

Updates the decipher with `data`. If the `inputEncoding` argument is given, its value must be one of `'latin1'`, `'base64'`, or `'hex'` and the `data` argument is a string using the specified encoding. If the `inputEncoding` argument is not given, `data` must be a Buffer . If `data` is a Buffer then `inputEncoding` is ignored.

The `outputEncoding` specifies the output format of the enciphered data, and can be `'latin1'`, `'ascii'` or `'utf8'`. If the `outputEncoding` is specified, a string using the specified encoding is returned. If no `outputEncoding` is provided, a Buffer is returned.

The `decipher.update()` method can be called multiple times with new data until decipher.final() is called. Calling `decipher.update()` after decipher.final() will result in an error being thrown.

Class: DiffieHellman

Added in: v0.5.0

The `DiffieHellman` class is a utility for creating Diffie-Hellman key exchanges.

Instances of the `DiffieHellman` class can be created using the crypto.createDiffieHellman() function.

```
const crypto = require('crypto');
const assert = require('assert');

// Generate Alice's keys...
const alice = crypto.createDiffieHellman(2048);
const aliceKey = alice.generateKeys();

// Generate Bob's keys...
const bob = crypto.createDiffieHellman(alice.getPrime(), alice.getGenerator());
const bobKey = bob.generateKeys();

// Exchange and generate the secret...
const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

// OK
assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex'));
```

diffieHellman.computeSecret(otherPublicKey[, inputEncoding][, outputEncoding])

Added in: v0.5.0

- `otherPublicKey` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>
- `outputEncoding` <string>

Computes the shared secret using `otherPublicKey` as the other party's public key and returns the computed shared secret. The supplied key is

interpreted using the specified `inputEncoding`, and secret is encoded using specified `outputEncoding`. Encodings can be `'latin1'`, `'hex'`, or `'base64'`. If the `inputEncoding` is not provided, `otherPublicKey` is expected to be a Buffer, `TypedArray`, or `DataView`.

If `outputEncoding` is given a string is returned; otherwise, a Buffer is returned.

diffieHellman.generateKeys([encoding])

Added in: v0.5.0

- `encoding` <string>

Generates private and public Diffie-Hellman key values, and returns the public key in the specified `encoding`. This key should be transferred to the other party. Encoding can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a Buffer is returned.

diffieHellman.getGenerator([encoding])

Added in: v0.5.0

- `encoding` <string>

Returns the Diffie-Hellman generator in the specified `encoding`, which can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a Buffer is returned.

diffieHellman.getPrime([encoding])

Added in: v0.5.0

- `encoding` <string>

Returns the Diffie-Hellman prime in the specified `encoding`, which can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a Buffer is returned.

diffieHellman.getPrivateKey([encoding])

Added in: v0.5.0

- `encoding` <string>

Returns the Diffie-Hellman private key in the specified `encoding`, which can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a Buffer is returned.

diffieHellman.getPublicKey([encoding])

Added in: v0.5.0

- `encoding` <string>

Returns the Diffie-Hellman public key in the specified `encoding`, which can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a Buffer is returned.

diffieHellman.setPrivateKey(privateKey[, encoding])

Added in: v0.5.0

- `privateKey` <string> | <Buffer> | <TypedArray> | <DataView>
- `encoding` <string>

Sets the Diffie-Hellman private key. If the `encoding` argument is provided and is either `'latin1'`, `'hex'`, or `'base64'`, `privateKey` is expected to be a string. If no `encoding` is provided, `privateKey` is expected to be a Buffer, `TypedArray`, or `DataView`.

diffieHellman.setPublicKey(publicKey[, encoding])

Added in: v0.5.0

- `publicKey` <string> | <Buffer> | <TypedArray> | <DataView>
- `encoding` <string>

Sets the Diffie-Hellman public key. If the `encoding` argument is provided and is either `'latin1'`, `'hex'` or `'base64'`, `publicKey` is expected to be a string. If no `encoding` is provided, `publicKey` is expected to be a Buffer, `TypedArray`, or `DataView`.

diffieHellman.verifyError

Added in: v0.11.12

A bit field containing any warnings and/or errors resulting from a check performed during initialization of the `DiffieHellman` object.

The following values are valid for this property (as defined in `constants` module):

- `DH_CHECK_P_NOT_SAFE_PRIME`
- `DH_CHECK_P_NOT_PRIME`
- `DH_UNABLE_TO_CHECK_GENERATOR`
- `DH_NOT_SUITABLE_GENERATOR`

Class: ECDH

Added in: v0.11.14

The `ECDH` class is a utility for creating Elliptic Curve Diffie-Hellman (ECDH) key exchanges.

Instances of the `ECDH` class can be created using the `crypto.createECDH()` function.

```
const crypto = require('crypto');
const assert = require('assert');

// Generate Alice's keys...
const alice = crypto.createECDH('secp521r1');
const aliceKey = alice.generateKeys();

// Generate Bob's keys...
const bob = crypto.createECDH('secp521r1');
const bobKey = bob.generateKeys();

// Exchange and generate the secret...
const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex'));
// OK
```

ecdh.computeSecret(otherPublicKey[, inputEncoding][, outputEncoding])

▸ History

- `otherPublicKey` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>
- `outputEncoding` <string>

Computes the shared secret using `otherPublicKey` as the other party's public key and returns the computed shared secret. The supplied key is interpreted using specified `inputEncoding`, and the returned secret is encoded using the specified `outputEncoding`. Encodings can be `'latin1'`, `'hex'`, or `'base64'`. If the `inputEncoding` is not provided, `otherPublicKey` is expected to be a Buffer, `TypedArray`, or `DataView`.

If `outputEncoding` is given a string will be returned; otherwise a Buffer is returned.

ecdh.generateKeys([encoding[, format]])

Added in: v0.11.14

- `encoding` <string>
- `format` <string> Defaults to `uncompressed`.

Generates private and public EC Diffie-Hellman key values, and returns the public key in the specified `format` and `encoding`. This key should be transferred to the other party.

The `format` argument specifies point encoding and can be `'compressed'` or `'uncompressed'`. If `format` is not specified, the point will be returned in `'uncompressed'` format.

The `encoding` argument can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a Buffer is returned.

ecdh.getPrivateKey([encoding])

Added in: v0.11.14

- `encoding` <string>

Returns the EC Diffie-Hellman private key in the specified `encoding`, which can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a Buffer is returned.

ecdh.getPublicKey([encoding][, format])

Added in: v0.11.14

- `encoding` <string>
- `format` <string> Defaults to `uncompressed`.

Returns the EC Diffie-Hellman public key in the specified `encoding` and `format`.

The `format` argument specifies point encoding and can be `'compressed'` or `'uncompressed'`. If `format` is not specified the point will be returned in `'uncompressed'` format.

The `encoding` argument can be `'latin1'`, `'hex'`, or `'base64'`. If `encoding` is specified, a string is returned; otherwise a Buffer is returned.

ecdh.setPrivateKey(privateKey[, encoding])

Added in: v0.11.14

- `privateKey` <string> | <Buffer> | <TypedArray> | <DataView>
- `encoding` <string>

Sets the EC Diffie-Hellman private key. The `encoding` can be `'latin1'`, `'hex'` or `'base64'`. If `encoding` is provided, `privateKey` is expected to be a string; otherwise `privateKey` is expected to be a Buffer, `TypedArray`, or `DataView`.

If `privateKey` is not valid for the curve specified when the `ECDH` object was created, an error is thrown. Upon setting the private key, the associated public point (key) is also generated and set in the ECDH object.

ecdh.setPublicKey(publicKey[, encoding])

Added in: v0.11.14 Deprecated since: v5.2.0

Stability: 0 - Deprecated

- `publicKey` <string> | <Buffer> | <TypedArray> | <DataView>
- `encoding` <string>

Sets the EC Diffie-Hellman public key. Key encoding can be `'latin1'`, `'hex'` or `'base64'`. If `encoding` is provided `publicKey` is expected to be a string; otherwise a Buffer, `TypedArray`, or `DataView` is expected.

Note that there is not normally a reason to call this method because `ECDH` only requires a private key and the other party's public key to compute the shared secret. Typically either ecdh.generateKeys() or ecdh.setPrivateKey() will be called. The ecdh.setPrivateKey() method attempts to generate the public point/key associated with the private key being set.

Example (obtaining a shared secret):

```
const crypto = require('crypto');
const alice = crypto.createECDH('secp256k1');
const bob = crypto.createECDH('secp256k1');

// Note: This is a shortcut way to specify one of Alice's previous private
// keys. It would be unwise to use such a predictable private key in a real
// application.
alice.setPrivateKey(
  crypto.createHash('sha256').update('alice', 'utf8').digest()
);

// Bob uses a newly generated cryptographically strong
// pseudorandom key pair
bob.generateKeys();

const aliceSecret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bobSecret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

// aliceSecret and bobSecret should be the same shared secret value
console.log(aliceSecret === bobSecret);
```

Class: Hash

Added in: v0.1.92

The `Hash` class is a utility for creating hash digests of data. It can be used in one of two ways:

- As a stream that is both readable and writable, where data is written to produce a computed hash digest on the readable side, or
- Using the hash.update() and hash.digest() methods to produce the computed hash.

The crypto.createHash() method is used to create `Hash` instances. `Hash` objects are not to be created directly using the `new` keyword.

Example: Using `Hash` objects as streams:

```
const crypto = require('crypto');
const hash = crypto.createHash('sha256');

hash.on('readable', () => {
  const data = hash.read();
  if (data) {
    console.log(data.toString('hex'));
    // Prints:
    //   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50
  }
});

hash.write('some data to hash');
hash.end();
```

Example: Using `Hash` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const hash = crypto.createHash('sha256');

const input = fs.createReadStream('test.js');
input.pipe(hash).pipe(process.stdout);
```

Example: Using the `hash.update()` and `hash.digest()` methods:

```
const crypto = require('crypto');
const hash = crypto.createHash('sha256');

hash.update('some data to hash');
console.log(hash.digest('hex'));
// Prints:
//   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50
```

hash.digest([encoding])

Added in: v0.1.92

- `encoding` <string>

Calculates the digest of all of the data passed to be hashed (using the `hash.update()` method). The `encoding` can be `'hex'`, `'latin1'` or `'base64'`. If `encoding` is provided a string will be returned; otherwise a `Buffer` is returned.

The `Hash` object can not be used again after `hash.digest()` method has been called. Multiple calls will cause an error to be thrown.

hash.update(data[, inputEncoding])

▸ History

- `data` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>

Updates the hash content with the given `data`, the encoding of which is given in `inputEncoding` and can be `'utf8'`, `'ascii'` or `'latin1'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

Class: Hmac

Added in: v0.1.94

The `Hmac` Class is a utility for creating cryptographic HMAC digests. It can be used in one of two ways:

- As a stream that is both readable and writable, where data is written to produce a computed HMAC digest on the readable side, or
- Using the `hmac.update()` and `hmac.digest()` methods to produce the computed HMAC digest.

The `crypto.createHmac()` method is used to create `Hmac` instances. `Hmac` objects are not to be created directly using the `new` keyword.

Example: Using `Hmac` objects as streams:

```
const crypto = require('crypto');
const hmac = crypto.createHmac('sha256', 'a secret');

hmac.on('readable', () => {
  const data = hmac.read();
  if (data) {
    console.log(data.toString('hex'));
    // Prints:
    //   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
  }
});

hmac.write('some data to hash');
hmac.end();
```

Example: Using `Hmac` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const hmac = crypto.createHmac('sha256', 'a secret');

const input = fs.createReadStream('test.js');
input.pipe(hmac).pipe(process.stdout);
```

Example: Using the `hmac.update()` and `hmac.digest()` methods:

```
const crypto = require('crypto');
const hmac = crypto.createHmac('sha256', 'a secret');

hmac.update('some data to hash');
console.log(hmac.digest('hex'));
// Prints:
//   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
```

hmac.digest([encoding])

Added in: v0.1.94

- `encoding` <string>

Calculates the HMAC digest of all of the data passed using `hmac.update()`. The `encoding` can be `'hex'`, `'latin1'` or `'base64'`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned;

The `Hmac` object can not be used again after `hmac.digest()` has been called. Multiple calls to `hmac.digest()` will result in an error being thrown.

hmac.update(data[, inputEncoding])

▸ History

- `data` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>

Updates the `Hmac` content with the given `data`, the encoding of which is given in `inputEncoding` and can be `'utf8'`, `'ascii'` or `'latin1'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

Class: Sign

Added in: v0.1.92

The `Sign` Class is a utility for generating signatures. It can be used in one of two ways:

- As a writable stream, where data to be signed is written and the sign.sign() method is used to generate and return the signature, or
- Using the sign.update() and sign.sign() methods to produce the signature.

The crypto.createSign() method is used to create `Sign` instances. The argument is the string name of the hash function to use. `Sign` objects are not to be created directly using the `new` keyword.

Example: Using `Sign` objects as streams:

```
const crypto = require('crypto');
const sign = crypto.createSign('SHA256');

sign.write('some data to sign');
sign.end();

const privateKey = getPrivateKeySomehow();
console.log(sign.sign(privateKey, 'hex'));
// Prints: the calculated signature using the specified private key and
// SHA-256. For RSA keys, the algorithm is RSASSA-PKCS1-v1_5 (see padding
// parameter below for RSASSA-PSS). For EC keys, the algorithm is ECDSA.
```

Example: Using the sign.update() and sign.sign() methods:

```
const crypto = require('crypto');
const sign = crypto.createSign('SHA256');

sign.update('some data to sign');

const privateKey = getPrivateKeySomehow();
console.log(sign.sign(privateKey, 'hex'));
// Prints: the calculated signature
```

In some cases, a `Sign` instance can also be created by passing in a signature algorithm name, such as 'RSA-SHA256'. This will use the corresponding digest algorithm. This does not work for all signature algorithms, such as 'ecdsa-with-SHA256'. Use digest names instead.

Example: signing using legacy signature algorithm name

```
const crypto = require('crypto');
const sign = crypto.createSign('RSA-SHA256');

sign.update('some data to sign');

const privateKey = getPrivateKeySomehow();
console.log(sign.sign(privateKey, 'hex'));
// Prints: the calculated signature
```

sign.sign(privateKey[, outputFormat])

▶ History

- `privateKey` <string> | <Object>
  - `key` <string>

    - `passphrase` <string>
- `outputFormat` <string>

Calculates the signature on all the data passed through using either `sign.update()` or `sign.write()`.

The `privateKey` argument can be an object or a string. If `privateKey` is a string, it is treated as a raw key with no passphrase. If `privateKey` is an object, it must contain one or more of the following properties:

- `key`: <string> - PEM encoded private key (required)
- `passphrase`: <string> - passphrase for the private key
- `padding`: <integer> - Optional padding value for RSA, one of the following:
    - `crypto.constants.RSA_PKCS1_PADDING` (default)
    - `crypto.constants.RSA_PKCS1_PSS_PADDING`

  Note that `RSA_PKCS1_PSS_PADDING` will use MGF1 with the same hash function used to sign the message as specified in section 3.1 of RFC 4055.
- `saltLength`: <integer> - salt length for when padding is `RSA_PKCS1_PSS_PADDING`. The special value `crypto.constants.RSA_PSS_SALTLEN_DIGEST` sets the salt length to the digest size, `crypto.constants.RSA_PSS_SALTLEN_MAX_SIGN` (default) sets it to the maximum permissible value.

The `outputFormat` can specify one of `'latin1'`, `'hex'` or `'base64'`. If `outputFormat` is provided a string is returned; otherwise a `Buffer` is returned.

The `Sign` object can not be again used after `sign.sign()` method has been called. Multiple calls to `sign.sign()` will result in an error being thrown.

sign.update(data[, inputEncoding])

▸ History

- `data` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>

Updates the `Sign` content with the given `data`, the encoding of which is given in `inputEncoding` and can be `'utf8'`, `'ascii'` or `'latin1'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

Class: Verify

Added in: v0.1.92

The `Verify` class is a utility for verifying signatures. It can be used in one of two ways:

- As a writable stream where written data is used to validate against the supplied signature, or
- Using the `verify.update()` and `verify.verify()` methods to verify the signature.

The `crypto.createVerify()` method is used to create `Verify` instances. `Verify` objects are not to be created directly using the `new` keyword.

Example: Using `Verify` objects as streams:

```
const crypto = require('crypto');
const verify = crypto.createVerify('SHA256');

verify.write('some data to sign');
verify.end();

const publicKey = getPublicKeySomehow();
const signature = getSignatureToVerify();
console.log(verify.verify(publicKey, signature));
// Prints: true or false
```

Example: Using the `verify.update()` and `verify.verify()` methods:

```
const crypto = require('crypto');
const verify = crypto.createVerify('SHA256');

verify.update('some data to sign');

const publicKey = getPublicKeySomehow();
const signature = getSignatureToVerify();
console.log(verify.verify(publicKey, signature));
// Prints: true or false
```

verify.update(data[, inputEncoding])

▸ History

- `data` <string> | <Buffer> | <TypedArray> | <DataView>
- `inputEncoding` <string>

Updates the `Verify` content with the given `data`, the encoding of which is given in `inputEncoding` and can be `'utf8'`, `'ascii'` or `'latin1'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

verify.verify(object, signature[, signatureFormat])

▸ History

- `object` <string> | <Object>
- `signature` <string> | <Buffer> | <TypedArray> | <DataView>
- `signatureFormat` <string>

Verifies the provided data using the given `object` and `signature`. The `object` argument can be either a string containing a PEM encoded object, which can be an RSA public key, a DSA public key, or an X.509 certificate, or an object with one or more of the following properties:

- `key`: <string> - PEM encoded public key (required)
- `padding`: <integer> - Optional padding value for RSA, one of the following:
  - `crypto.constants.RSA_PKCS1_PADDING` (default)
  - `crypto.constants.RSA_PKCS1_PSS_PADDING`

  Note that `RSA_PKCS1_PSS_PADDING` will use MGF1 with the same hash function used to verify the message as specified in section 3.1 of RFC 4055.
- `saltLength`: <integer> - salt length for when padding is `RSA_PKCS1_PSS_PADDING`. The special value `crypto.constants.RSA_PSS_SALTLEN_DIGEST` sets the salt length to the digest size, `crypto.constants.RSA_PSS_SALTLEN_AUTO` (default)

causes it to be determined automatically.

The `signature` argument is the previously calculated signature for the data, in the `signatureFormat` which can be `'latin1'`, `'hex'` or `'base64'`. If a `signatureFormat` is specified, the `signature` is expected to be a string; otherwise `signature` is expected to be a Buffer, `TypedArray`, or `DataView`.

Returns `true` or `false` depending on the validity of the signature for the data and public key.

The `verify` object can not be used again after `verify.verify()` has been called. Multiple calls to `verify.verify()` will result in an error being thrown.

`crypto` module methods and properties

crypto.constants

Added in: v6.3.0

Returns an object containing commonly used constants for crypto and security related operations. The specific constants currently defined are described in Crypto Constants.

crypto.DEFAULT_ENCODING

Added in: v0.9.3

The default encoding to use for functions that can take either strings or buffers. The default value is `'buffer'`, which makes methods default to Buffer objects.

The `crypto.DEFAULT_ENCODING` mechanism is provided for backwards compatibility with legacy programs that expect `'latin1'` to be the default encoding.

New applications should expect the default to be `'buffer'`. This property may become deprecated in a future Node.js release.

crypto.fips

Added in: v6.0.0

Property for checking and controlling whether a FIPS compliant crypto provider is currently in use. Setting to true requires a FIPS build of Node.js.

crypto.createCipher(algorithm, password[, options])

Added in: v0.1.94

- `algorithm` <string>
- `password` <string> | <Buffer> | <TypedArray> | <DataView>
- `options` <Object> stream.transform options

Creates and returns a `Cipher` object that uses the given `algorithm` and `password`. Optional `options` argument controls stream behavior.

The `algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent OpenSSL releases, `openssl list-cipher-algorithms` will display the available cipher algorithms.

The `password` is used to derive the cipher key and initialization vector (IV). The value must be either a `'latin1'` encoded string, a Buffer, a `TypedArray`, or a `DataView`.

The implementation of `crypto.createCipher()` derives keys using the OpenSSL function EVP_BytesToKey with the digest algorithm set to MD5, one iteration, and no salt. The lack of salt allows dictionary attacks as the same password always creates the same key. The low iteration count and non-cryptographically secure hash algorithm allow passwords to be tested very rapidly.

In line with OpenSSL's recommendation to use PBKDF2 instead of EVP_BytesToKey it is recommended that developers derive a key and IV on their own using crypto.pbkdf2() and to use crypto.createCipheriv() to create the `Cipher` object. Users should not use ciphers with counter mode (e.g. CTR, GCM, or CCM) in `crypto.createCipher()`. A warning is emitted when they are used in order to avoid the risk of IV reuse that causes vulnerabilities. For the case when IV is reused in GCM, see Nonce-Disrespecting Adversaries for details.

crypto.createCipheriv(algorithm, key, iv[, options])

▸ History

- `algorithm` <string>
- `key` <string> | <Buffer> | <TypedArray> | <DataView>
- `iv` <string> | <Buffer> | <TypedArray> | <DataView>
- `options` <Object> stream.transform options

Creates and returns a `Cipher` object, with the given `algorithm`, `key` and initialization vector (`iv`). Optional `options` argument controls stream behavior.

The `algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent OpenSSL releases, `openssl list-cipher-algorithms` will display the available cipher algorithms.

The `key` is the raw key used by the `algorithm` and `iv` is an initialization vector. Both arguments must be `'utf8'` encoded strings, Buffers, `TypedArray`, or `DataView`s. If the cipher does not need an initialization vector, `iv` may be `null`.

crypto.createCredentials(details)

Added in: v0.1.92 Deprecated since: v0.11.13

Stability: 0 - Deprecated: Use tls.createSecureContext() instead.

- `details` <Object> Identical to tls.createSecureContext().

The `crypto.createCredentials()` method is a deprecated function for creating and returning a `tls.SecureContext`. It should not be used. Replace it with tls.createSecureContext() which has the exact same arguments and return value.

Returns a `tls.SecureContext`, as-if tls.createSecureContext() had been called.

crypto.createDecipher(algorithm, password[, options])

Added in: v0.1.94

- `algorithm` <string>
- `password` <string> | <Buffer> | <TypedArray> | <DataView>
- `options` <Object> stream.transform options

Creates and returns a `Decipher` object that uses the given `algorithm` and `password` (key). Optional `options` argument controls stream behavior.

The implementation of `crypto.createDecipher()` derives keys using the OpenSSL function EVP_BytesToKey with the digest algorithm set to MD5, one iteration, and no salt. The lack of salt allows dictionary attacks as the same password always creates the same key. The low iteration count and non-cryptographically secure hash algorithm allow passwords to be tested very rapidly.

In line with OpenSSL's recommendation to use PBKDF2 instead of EVP_BytesToKey it is recommended that developers derive a key and IV on their own using crypto.pbkdf2() and to use crypto.createDecipheriv() to create the `Decipher` object.

crypto.createDecipheriv(algorithm, key, iv[, options])

▸ History

- `algorithm` <string>
- `key` <string> | <Buffer> | <TypedArray> | <DataView>
- `iv` <string> | <Buffer> | <TypedArray> | <DataView>
- `options` <Object> stream.transform options

Creates and returns a `Decipher` object that uses the given `algorithm`, `key` and initialization vector (`iv`). Optional `options` argument controls stream behavior.

The `algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent OpenSSL releases, `openssl list-cipher-algorithms` will display the available cipher algorithms.

The `key` is the raw key used by the `algorithm` and `iv` is an initialization vector. Both arguments must be `'utf8'` encoded strings, Buffers, `TypedArray`, or `DataView`s. If the cipher does not need an initialization vector, `iv` may be `null`.

crypto.createDiffieHellman(prime[, primeEncoding][, generator][, generatorEncoding])

▸ History

- `prime` <string> | <Buffer> | <TypedArray> | <DataView>
- `primeEncoding` <string>
- `generator` <number> | <string> | <Buffer> | <TypedArray> | <DataView> Defaults to `2`.
- `generatorEncoding` <string>

Creates a `DiffieHellman` key exchange object using the supplied `prime` and an optional specific `generator`.

The `generator` argument can be a number, string, or `Buffer`. If `generator` is not specified, the value `2` is used.

The `primeEncoding` and `generatorEncoding` arguments can be `'latin1'`, `'hex'`, or `'base64'`.

If `primeEncoding` is specified, `prime` is expected to be a string; otherwise a `Buffer`, `TypedArray`, or `DataView` is expected.

If `generatorEncoding` is specified, `generator` is expected to be a string; otherwise a number, `Buffer`, `TypedArray`, or `DataView` is expected.

crypto.createDiffieHellman(primeLength[, generator])

Added in: v0.5.0

- `primeLength` <number>
- `generator` <number> | <string> | <Buffer> | <TypedArray> | <DataView> Defaults to `2`.

Creates a `DiffieHellman` key exchange object and generates a prime of `primeLength` bits using an optional specific numeric `generator`. If `generator` is not specified, the value `2` is used.

crypto.createECDH(curveName)

Added in: v0.11.14

- `curveName` <string>

Creates an Elliptic Curve Diffie-Hellman (`ECDH`) key exchange object using a predefined curve specified by the `curveName` string. Use `crypto.getCurves()` to obtain a list of available curve names. On recent OpenSSL releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve.

crypto.createHash(algorithm[, options])

Added in: v0.1.92

- `algorithm` <string>
- `options` <Object> `stream.transform` options

Creates and returns a `Hash` object that can be used to generate hash digests using the given `algorithm`. Optional `options` argument controls stream behavior.

The `algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha256'`, `'sha512'`, etc. On recent releases of OpenSSL, `openssl list-message-digest-algorithms` will display the available digest algorithms.

Example: generating the sha256 sum of a file

```
const filename = process.argv[2];
const crypto = require('crypto');
const fs = require('fs');

const hash = crypto.createHash('sha256');

const input = fs.createReadStream(filename);
input.on('readable', () => {
  const data = input.read();
  if (data)
    hash.update(data);
  else {
    console.log(`${hash.digest('hex')} ${filename}`);
  }
});
```

crypto.createHmac(algorithm, key[, options])

Added in: v0.1.94

- `algorithm` <string>
- `key` <string> | <Buffer> | <TypedArray> | <DataView>
- `options` <Object> stream.transform options

Creates and returns an `Hmac` object that uses the given `algorithm` and `key`. Optional `options` argument controls stream behavior.

The `algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha256'`, `'sha512'`, etc. On recent releases of OpenSSL, `openssl list-message-digest-algorithms` will display the available digest algorithms.

The `key` is the HMAC key used to generate the cryptographic HMAC hash.

Example: generating the sha256 HMAC of a file

```
const filename = process.argv[2];
const crypto = require('crypto');
const fs = require('fs');

const hmac = crypto.createHmac('sha256', 'a secret');

const input = fs.createReadStream(filename);
input.on('readable', () => {
  const data = input.read();
  if (data)
    hmac.update(data);
  else {
    console.log(`${hmac.digest('hex')} ${filename}`);
  }
});
```

crypto.createSign(algorithm[, options])

Added in: v0.1.92

- `algorithm` <string>
- `options` <Object> stream.Writable options

Creates and returns a `Sign` object that uses the given `algorithm`. Use crypto.getHashes() to obtain an array of names of the available signing

algorithms. Optional `options` argument controls the `stream.Writable` behavior.

crypto.createVerify(algorithm[, options])

Added in: v0.1.92

- `algorithm` <string>
- `options` <Object> stream.Writable options

Creates and returns a `Verify` object that uses the given algorithm. Use crypto.getHashes() to obtain an array of names of the available signing algorithms. Optional `options` argument controls the `stream.Writable` behavior.

crypto.getCiphers()

Added in: v0.9.3

Returns an array with the names of the supported cipher algorithms.

Example:

```
const ciphers = crypto.getCiphers();
console.log(ciphers); // ['aes-128-cbc', 'aes-128-ccm', ...]
```

crypto.getCurves()

Added in: v2.3.0

Returns an array with the names of the supported elliptic curves.

Example:

```
const curves = crypto.getCurves();
console.log(curves); // ['Oakley-EC2N-3', 'Oakley-EC2N-4', ...]
```

crypto.getDiffieHellman(groupName)

Added in: v0.7.5

- `groupName` <string>

Creates a predefined `DiffieHellman` key exchange object. The supported groups are: `'modp1'`, `'modp2'`, `'modp5'` (defined in RFC 2412, but see Caveats) and `'modp14'`, `'modp15'`, `'modp16'`, `'modp17'`, `'modp18'` (defined in RFC 3526). The returned object mimics the interface of objects created by crypto.createDiffieHellman(), but will not allow changing the keys (with diffieHellman.setPublicKey() for example). The advantage of using this method is that the parties do not have to generate nor exchange a group modulus beforehand, saving both processor and communication time.

Example (obtaining a shared secret):

```
const crypto = require('crypto');
const alice = crypto.getDiffieHellman('modp14');
const bob = crypto.getDiffieHellman('modp14');

alice.generateKeys();
bob.generateKeys();

const aliceSecret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bobSecret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

/* aliceSecret and bobSecret should be the same */
console.log(aliceSecret === bobSecret);
```

crypto.getHashes()

Added in: v0.9.3

Returns an array of the names of the supported hash algorithms, such as `RSA-SHA256`.

Example:

```
const hashes = crypto.getHashes();
console.log(hashes); // ['DSA', 'DSA-SHA', 'DSA-SHA1', ...]
```

crypto.pbkdf2(password, salt, iterations, keylen, digest, callback)

▸ History

- `password` <string> | <Buffer> | <TypedArray>
- `salt` <string> | <Buffer> | <TypedArray>
- `iterations` <number>
- `keylen` <number>
- `digest` <string>
- `callback` <Function>
  - `err` <Error>
  - `derivedKey` <Buffer>

Provides an asynchronous Password-Based Key Derivation Function 2 (PBKDF2) implementation. A selected HMAC digest algorithm specified by `digest` is applied to derive a key of the requested byte length (`keylen`) from the `password`, `salt` and `iterations`.

The supplied `callback` function is called with two arguments: `err` and `derivedKey`. If an error occurs while deriving the key, `err` will be set; otherwise `err` will be null. By default, the successfully generated `derivedKey` will be passed to the callback as a Buffer. An error will be thrown if any of the input arguments specify invalid values or types.

The `iterations` argument must be a number set as high as possible. The higher the number of iterations, the more secure the derived key will be, but will take a longer amount of time to complete.

The `salt` should also be as unique as possible. It is recommended that the salts are random and their lengths are at least 16 bytes. See NIST SP 800-132 for details.

Example:

```
const crypto = require('crypto');
crypto.pbkdf2('secret', 'salt', 100000, 64, 'sha512', (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey.toString('hex'));  // '3745e48...08d59ae'
});
```

The `crypto.DEFAULT_ENCODING` may be used to change the way the `derivedKey` is passed to the callback:

```
const crypto = require('crypto');
crypto.DEFAULT_ENCODING = 'hex';
crypto.pbkdf2('secret', 'salt', 100000, 512, 'sha512', (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey);  // '3745e48...aa39b34'
});
```

An array of supported digest functions can be retrieved using crypto.getHashes().

Note that this API uses libuv's threadpool, which can have surprising and negative performance implications for some applications, see the UV_THREADPOOL_SIZE documentation for more information.

crypto.pbkdf2Sync(password, salt, iterations, keylen, digest)

▸ History

- `password` <string> | <Buffer> | <TypedArray>
- `salt` <string> | <Buffer> | <TypedArray>
- `iterations` <number>
- `keylen` <number>
- `digest` <string>

Provides a synchronous Password-Based Key Derivation Function 2 (PBKDF2) implementation. A selected HMAC digest algorithm specified by `digest` is applied to derive a key of the requested byte length (`keylen`) from the `password`, `salt` and `iterations`.

If an error occurs an Error will be thrown, otherwise the derived key will be returned as a Buffer.

The `iterations` argument must be a number set as high as possible. The higher the number of iterations, the more secure the derived key will be, but will take a longer amount of time to complete.

The `salt` should also be as unique as possible. It is recommended that the salts are random and their lengths are at least 16 bytes. See NIST SP 800-132 for details.

Example:

```
const crypto = require('crypto');
const key = crypto.pbkdf2Sync('secret', 'salt', 100000, 64, 'sha512');
console.log(key.toString('hex'));  // '3745e48...08d59ae'
```

The `crypto.DEFAULT_ENCODING` property may be used to change the way the `derivedKey` is returned.

```
const crypto = require('crypto');
crypto.DEFAULT_ENCODING = 'hex';
const key = crypto.pbkdf2Sync('secret', 'salt', 100000, 512, 'sha512');
console.log(key);  // '3745e48...aa39b34'
```

An array of supported digest functions can be retrieved using crypto.getHashes().

crypto.privateDecrypt(privateKey, buffer)

Added in: v0.11.14

- `privateKey` <Object> | <string>
  - `key` <string> A PEM encoded private key.
  - `passphrase` <string> An optional passphrase for the private key.
  - `padding` <crypto.constants> An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING`, `RSA_PKCS1_PADDING`, or `crypto.constants.RSA_PKCS1_OAEP_PADDING`.
- `buffer` <Buffer> | <TypedArray> | <DataView>
- Returns: <Buffer> A new `Buffer` with the decrypted content.

Decrypts `buffer` with `privateKey`.

`privateKey` can be an object or a string. If `privateKey` is a string, it is treated as the key with no passphrase and will use `RSA_PKCS1_OAEP_PADDING`.

crypto.privateEncrypt(privateKey, buffer)

Added in: v1.1.0

- `privateKey` <Object> | <string>
  - `key` <string> A PEM encoded private key.
  - `passphrase` <string> An optional passphrase for the private key.
  - `padding` <crypto.constants> An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING` or `RSA_PKCS1_PADDING`.
- `buffer` <Buffer> | <TypedArray> | <DataView>
- Returns: <Buffer> A new `Buffer` with the encrypted content.

Encrypts `buffer` with `privateKey`.

`privateKey` can be an object or a string. If `privateKey` is a string, it is treated as the key with no passphrase and will use `RSA_PKCS1_PADDING`.

crypto.publicDecrypt(key, buffer)

Added in: v1.1.0

- `key` <Object> | <string>
  - `key` <string> A PEM encoded public or private key.
  - `passphrase` <string> An optional passphrase for the private key.
  - `padding` <crypto.constants> An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING` or `RSA_PKCS1_PADDING`.
- `buffer` <Buffer> | <TypedArray> | <DataView>
- Returns: <Buffer> A new `Buffer` with the decrypted content.

Decrypts `buffer` with `key`.

`key` can be an object or a string. If `key` is a string, it is treated as the key with no passphrase and will use `RSA_PKCS1_PADDING`.

Because RSA public keys can be derived from private keys, a private key may be passed instead of a public key.

crypto.publicEncrypt(key, buffer)

Added in: v0.11.14

- `key` <Object> | <string>
  - `key` <string> A PEM encoded public or private key.
  - `passphrase` <string> An optional passphrase for the private key.
  - `padding` <crypto.constants> An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING`, `RSA_PKCS1_PADDING`, or `crypto.constants.RSA_PKCS1_OAEP_PADDING`.

- buffer <Buffer> | <TypedArray> | <DataView>
- Returns: <Buffer> A new Buffer with the encrypted content.

Encrypts the content of buffer with key and returns a new Buffer with encrypted content.

key can be an object or a string. If key is a string, it is treated as the key with no passphrase and will use RSA_PKCS1_OAEP_PADDING.

Because RSA public keys can be derived from private keys, a private key may be passed instead of a public key.

crypto.randomBytes(size[, callback])

▸ History

- size <number>
- callback <Function>
  - err <Error>
  - buf <Buffer>

Generates cryptographically strong pseudo-random data. The size argument is a number indicating the number of bytes to generate.

If a callback function is provided, the bytes are generated asynchronously and the callback function is invoked with two arguments: err and buf. If an error occurs, err will be an Error object; otherwise it is null. The buf argument is a Buffer containing the generated bytes.

```
// Asynchronous
const crypto = require('crypto');
crypto.randomBytes(256, (err, buf) => {
  if (err) throw err;
  console.log(`${buf.length} bytes of random data: ${buf.toString('hex')}`);
});
```

If the callback function is not provided, the random bytes are generated synchronously and returned as a Buffer. An error will be thrown if there is a problem generating the bytes.

```
// Synchronous
const buf = crypto.randomBytes(256);
console.log(
  `${buf.length} bytes of random data: ${buf.toString('hex')}`);
```

The crypto.randomBytes() method will not complete until there is sufficient entropy available. This should normally never take longer than a few milliseconds. The only time when generating the random bytes may conceivably block for a longer period of time is right after boot, when the whole system is still low on entropy.

Note that this API uses libuv's threadpool, which can have surprising and negative performance implications for some applications, see the UV_THREADPOOL_SIZE documentation for more information.

The asynchronous version of crypto.randomBytes() is carried out in a single threadpool request. To minimize threadpool task length variation, partition large randomBytes requests when doing so as part of fulfilling a client request.

crypto.randomFillSync(buffer[, offset][, size])

▸ History

- buffer <Buffer> | <TypedArray> | <DataView> Must be supplied.
- offset <number> Defaults to 0.
- size <number> Defaults to buffer.length - offset.

Synchronous version of crypto.randomFill().

Returns buffer

```
const buf = Buffer.alloc(10);
console.log(crypto.randomFillSync(buf).toString('hex'));

crypto.randomFillSync(buf, 5);
console.log(buf.toString('hex'));

// The above is equivalent to the following:
crypto.randomFillSync(buf, 5, 5);
console.log(buf.toString('hex'));
```

Any `TypedArray` or `DataView` instance may be passed as `buffer`.

```
const a = new Uint32Array(10);
console.log(crypto.randomFillSync(a).toString('hex'));

const b = new Float64Array(10);
console.log(crypto.randomFillSync(a).toString('hex'));

const c = new DataView(new ArrayBuffer(10));
console.log(crypto.randomFillSync(a).toString('hex'));
```

crypto.randomFill(buffer[, offset][, size], callback)

▸ History

- `buffer` <Buffer> | <TypedArray> | <DataView> Must be supplied.
- `offset` <number> Defaults to `0`.
- `size` <number> Defaults to `buffer.length - offset`.
- `callback` <Function> `function(err, buf) {}`.

This function is similar to `crypto.randomBytes()` but requires the first argument to be a `Buffer` that will be filled. It also requires that a callback is passed in.

If the `callback` function is not provided, an error will be thrown.

```
const buf = Buffer.alloc(10);
crypto.randomFill(buf, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

crypto.randomFill(buf, 5, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

// The above is equivalent to the following:
crypto.randomFill(buf, 5, 5, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});
```

Any `TypedArray` or `DataView` instance may be passed as `buffer`.

```
const a = new Uint32Array(10);
crypto.randomFill(a, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

const b = new Float64Array(10);
crypto.randomFill(b, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

const c = new DataView(new ArrayBuffer(10));
crypto.randomFill(c, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});
```

Note that this API uses libuv's threadpool, which can have surprising and negative performance implications for some applications, see the UV_THREADPOOL_SIZE documentation for more information.

The asynchronous version of `crypto.randomFill()` is carried out in a single threadpool request. To minimize threadpool task length variation, partition large `randomFill` requests when doing so as part of fulfilling a client request.

crypto.setEngine(engine[, flags])

Added in: v0.11.11

- `engine` <string>
- `flags` <crypto.constants> Defaults to `crypto.constants.ENGINE_METHOD_ALL`.

Load and set the `engine` for some or all OpenSSL functions (selected by flags).

`engine` could be either an id or a path to the engine's shared library.

The optional `flags` argument uses `ENGINE_METHOD_ALL` by default. The `flags` is a bit field taking one of or a mix of the following flags (defined in `crypto.constants`):

- `crypto.constants.ENGINE_METHOD_RSA`
- `crypto.constants.ENGINE_METHOD_DSA`
- `crypto.constants.ENGINE_METHOD_DH`
- `crypto.constants.ENGINE_METHOD_RAND`
- `crypto.constants.ENGINE_METHOD_ECDH`
- `crypto.constants.ENGINE_METHOD_ECDSA`
- `crypto.constants.ENGINE_METHOD_CIPHERS`
- `crypto.constants.ENGINE_METHOD_DIGESTS`
- `crypto.constants.ENGINE_METHOD_STORE`
- `crypto.constants.ENGINE_METHOD_PKEY_METHS`
- `crypto.constants.ENGINE_METHOD_PKEY_ASN1_METHS`
- `crypto.constants.ENGINE_METHOD_ALL`
- `crypto.constants.ENGINE_METHOD_NONE`

crypto.timingSafeEqual(a, b)

Added in: v6.6.0

- `a` <Buffer> | <TypedArray> | <DataView>
- `b` <Buffer> | <TypedArray> | <DataView>

This function is based on a constant-time algorithm. Returns true if `a` is equal to `b`, without leaking timing information that would allow an attacker to guess one of the values. This is suitable for comparing HMAC digests or secret values like authentication cookies or capability urls.

`a` and `b` must both be `Buffer`s, `TypedArray`s, or `DataView`s, and they must have the same length.

Use of `crypto.timingSafeEqual` does not guarantee that the *surrounding* code is timing-safe. Care should be taken to ensure that the surrounding code does not introduce timing vulnerabilities.

Notes

Legacy Streams API (pre Node.js v0.10)

The Crypto module was added to Node.js before there was the concept of a unified Stream API, and before there were Buffer objects for handling binary data. As such, the many of the `crypto` defined classes have methods not typically found on other Node.js classes that implement the streams API (e.g. `update()`, `final()`, or `digest()`). Also, many methods accepted and returned `'latin1'` encoded strings by default rather than Buffers. This default was changed after Node.js v0.8 to use Buffer objects by default instead.

Recent ECDH Changes

Usage of `ECDH` with non-dynamically generated key pairs has been simplified. Now, ecdh.setPrivateKey() can be called with a preselected private key and the associated public point (key) will be computed and stored in the object. This allows code to only store and provide the private part of the EC key pair. ecdh.setPrivateKey() now also validates that the private key is valid for the selected curve.

The ecdh.setPublicKey() method is now deprecated as its inclusion in the API is not useful. Either a previously stored private key should be set, which automatically generates the associated public key, or ecdh.generateKeys() should be called. The main drawback of using ecdh.setPublicKey() is that it can be used to put the ECDH key pair into an inconsistent state.

Support for weak or compromised algorithms

The `crypto` module still supports some algorithms which are already compromised and are not currently recommended for use. The API also allows the use of ciphers and hashes with a small key size that are considered to be too weak for safe use.

Users should take full responsibility for selecting the crypto algorithm and key size according to their security requirements.

Based on the recommendations of NIST SP 800-131A:

- MD5 and SHA-1 are no longer acceptable where collision resistance is required such as digital signatures.
- The key used with RSA, DSA, and DH algorithms is recommended to have at least 2048 bits and that of the curve of ECDSA and ECDH at least 224 bits, to be safe to use for several years.
- The DH groups of `modp1`, `modp2` and `modp5` have a key size smaller than 2048 bits and are not recommended.

See the reference for other recommendations and details.

Crypto Constants

The following constants exported by `crypto.constants` apply to various uses of the `crypto`, `tls`, and `https` modules and are generally specific to OpenSSL.

OpenSSL Options

Constant	Description
`SSL_OP_ALL`	Applies multiple bug workarounds within OpenSSL. See https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_set_options.html for detail.
`SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION`	Allows legacy insecure renegotiation between OpenSSL and unpatched

	clients or servers. See https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_set_options.html .
SSL_OP_CIPHER_SERVER_PREFERENCE	Attempts to use the server's preferences instead of the client's when selecting a cipher. Behavior depends on protocol version. See https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_set_options.html .
SSL_OP_CISCO_ANYCONNECT	Instructs OpenSSL to use Cisco's "speshul" version of DTLS_BAD_VER.
SSL_OP_COOKIE_EXCHANGE	Instructs OpenSSL to turn on cookie exchange.
SSL_OP_CRYPTOPRO_TLSEXT_BUG	Instructs OpenSSL to add server-hello extension from an early version of the cryptopro draft.
SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS	Instructs OpenSSL to disable a SSL 3.0/TLS 1.0 vulnerability workaround added in OpenSSL 0.9.6d.
SSL_OP_EPHEMERAL_RSA	Instructs OpenSSL to always use the tmp_rsa key when performing RSA operations.
SSL_OP_LEGACY_SERVER_CONNECT	Allows initial connection to servers that do not support RI.
SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER	
SSL_OP_MICROSOFT_SESS_ID_BUG	
SSL_OP_MSIE_SSLV2_RSA_PADDING	Instructs OpenSSL to disable the workaround for a man-in-the-middle protocol-version vulnerability in the SSL 2.0 server implementation.
SSL_OP_NETSCAPE_CA_DN_BUG	
SSL_OP_NETSCAPE_CHALLENGE_BUG	
SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG	
SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG	
SSL_OP_NO_COMPRESSION	Instructs OpenSSL to disable support for SSL/TLS compression.
SSL_OP_NO_QUERY_MTU	
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	Instructs OpenSSL to always start a new session when performing renegotiation.
SSL_OP_NO_SSLv2	Instructs OpenSSL to turn off SSL v2
SSL_OP_NO_SSLv3	Instructs OpenSSL to turn off SSL v3
SSL_OP_NO_TICKET	Instructs OpenSSL to disable use of RFC4507bis tickets.
SSL_OP_NO_TLSv1	Instructs OpenSSL to turn off TLS v1
SSL_OP_NO_TLSv1_1	Instructs OpenSSL to turn off TLS v1.1
SSL_OP_NO_TLSv1_2	Instructs OpenSSL to turn off TLS v1.2
SSL_OP_PKCS1_CHECK_1	
SSL_OP_PKCS1_CHECK_2	
SSL_OP_SINGLE_DH_USE	Instructs OpenSSL to always create a new key when using temporary/ephemeral DH parameters.

`SSL_OP_SINGLE_ECDH_USE`	Instructs OpenSSL to always create a new key when using temporary/ephemeral ECDH parameters.
`SSL_OP_SSLEAY_080_CLIENT_DH_BUG`	
`SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG`	
`SSL_OP_TLS_BLOCK_PADDING_BUG`	
`SSL_OP_TLS_D5_BUG`	
`SSL_OP_TLS_ROLLBACK_BUG`	Instructs OpenSSL to disable version rollback attack detection.

OpenSSL Engine Constants

Constant	Description
`ENGINE_METHOD_RSA`	Limit engine usage to RSA
`ENGINE_METHOD_DSA`	Limit engine usage to DSA
`ENGINE_METHOD_DH`	Limit engine usage to DH
`ENGINE_METHOD_RAND`	Limit engine usage to RAND
`ENGINE_METHOD_ECDH`	Limit engine usage to ECDH
`ENGINE_METHOD_ECDSA`	Limit engine usage to ECDSA
`ENGINE_METHOD_CIPHERS`	Limit engine usage to CIPHERS
`ENGINE_METHOD_DIGESTS`	Limit engine usage to DIGESTS
`ENGINE_METHOD_STORE`	Limit engine usage to STORE
`ENGINE_METHOD_PKEY_METHS`	Limit engine usage to PKEY_METHDS
`ENGINE_METHOD_PKEY_ASN1_METHS`	Limit engine usage to PKEY_ASN1_METHS
`ENGINE_METHOD_ALL`	
`ENGINE_METHOD_NONE`	

Other OpenSSL Constants

Constant	Description
`DH_CHECK_P_NOT_SAFE_PRIME`	
`DH_CHECK_P_NOT_PRIME`	
`DH_UNABLE_TO_CHECK_GENERATOR`	
`DH_NOT_SUITABLE_GENERATOR`	
`NPN_ENABLED`	

`ALPN_ENABLED`	
`RSA_PKCS1_PADDING`	
`RSA_SSLV23_PADDING`	
`RSA_NO_PADDING`	
`RSA_PKCS1_OAEP_PADDING`	
`RSA_X931_PADDING`	
`RSA_PKCS1_PSS_PADDING`	
`RSA_PSS_SALTLEN_DIGEST`	Sets the salt length for `RSA_PKCS1_PSS_PADDING` to the digest size when signing or verifying.
`RSA_PSS_SALTLEN_MAX_SIGN`	Sets the salt length for `RSA_PKCS1_PSS_PADDING` to the maximum permissible value when signing data.
`RSA_PSS_SALTLEN_AUTO`	Causes the salt length for `RSA_PKCS1_PSS_PADDING` to be determined automatically when verifying a signature.
`POINT_CONVERSION_COMPRESSED`	
`POINT_CONVERSION_UNCOMPRESSED`	
`POINT_CONVERSION_HYBRID`	

Node.js Crypto Constants

Constant	Description
`defaultCoreCipherList`	Specifies the built-in default cipher list used by Node.js.
`defaultCipherList`	Specifies the active default cipher list used by the current Node.js process.

Debugger

Stability: 2 - Stable

Node.js includes an out-of-process debugging utility accessible via a V8 Inspector and built-in debugging client. To use it, start Node.js with the `inspect` argument followed by the path to the script to debug; a prompt will be displayed indicating successful launch of the debugger:

```
$ node inspect myscript.js
< Debugger listening on ws://127.0.0.1:9229/80e7a814-7cd3-49fb-921a-2e02228cd5ba
< For help see https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in myscript.js:1
> 1 (function (exports, require, module, __filename, __dirname) { global.x = 5;
  2 setTimeout(() => {
  3   console.log('world');
debug>
```

Node.js's debugger client is not a full-featured debugger, but simple step and inspection are possible.

Inserting the statement `debugger;` into the source code of a script will enable a breakpoint at that position in the code:

```
// myscript.js
global.x = 5;
setTimeout(() => {
  debugger;
  console.log('world');
}, 1000);
console.log('hello');
```

Once the debugger is run, a breakpoint will occur at line 3:

```
$ node inspect myscript.js
< Debugger listening on ws://127.0.0.1:9229/80e7a814-7cd3-49fb-921a-2e02228cd5ba
< For help see https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in myscript.js:1
> 1 (function (exports, require, module, __filename, __dirname) { global.x = 5;
  2 setTimeout(() => {
  3   debugger;
debug> cont
< hello
break in myscript.js:3
  1 (function (exports, require, module, __filename, __dirname) { global.x = 5;
  2 setTimeout(() => {
> 3   debugger;
  4   console.log('world');
  5 }, 1000);
debug> next
break in myscript.js:4
  2 setTimeout(() => {
  3   debugger;
> 4   console.log('world');
  5 }, 1000);
  6 console.log('hello');
debug> repl
Press Ctrl + C to leave debug repl
> x
5
> 2 + 2
4
debug> next
< world
break in myscript.js:5
  3   debugger;
  4   console.log('world');
> 5 }, 1000);
  6 console.log('hello');
  7
debug> .exit
```

The `repl` command allows code to be evaluated remotely. The `next` command steps to the next line. Type `help` to see what other commands are available.

Pressing `enter` without typing a command will repeat the previous debugger command.

Watchers

It is possible to watch expression and variable values while debugging. On every breakpoint, each expression from the watchers list will be evaluated in the current context and displayed immediately before the breakpoint's source code listing.

To begin watching an expression, type `watch('my_expression')`. The command `watchers` will print the active watchers. To remove a watcher, type `unwatch('my_expression')`.

Command reference

Stepping

- `cont`, `c` - Continue execution
- `next`, `n` - Step next
- `step`, `s` - Step in
- `out`, `o` - Step out
- `pause` - Pause running code (like pause button in Developer Tools)

Breakpoints

- `setBreakpoint()`, `sb()` - Set breakpoint on current line
- `setBreakpoint(line)`, `sb(line)` - Set breakpoint on specific line
- `setBreakpoint('fn()')`, `sb(...)` - Set breakpoint on a first statement in functions body
- `setBreakpoint('script.js', 1)`, `sb(...)` - Set breakpoint on first line of script.js
- `clearBreakpoint('script.js', 1)`, `cb(...)` - Clear breakpoint in script.js on line 1

It is also possible to set a breakpoint in a file (module) that is not loaded yet:

```
$ node inspect main.js
< Debugger listening on ws://127.0.0.1:9229/4e3db158-9791-4274-8909-914f7facf3bd
< For help see https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in main.js:1
> 1 (function (exports, require, module, __filename, __dirname) { const mod = require('./mod.js');
  2 mod.hello();
  3 mod.hello();
debug> setBreakpoint('mod.js', 22)
Warning: script 'mod.js' was not loaded yet.
debug> c
break in mod.js:22
 20 // USE OR OTHER DEALINGS IN THE SOFTWARE.
 21
>22 exports.hello = function() {
 23   return 'hello from module';
 24 };
debug>
```

Information

- `backtrace`, `bt` - Print backtrace of current execution frame
- `list(5)` - List scripts source code with 5 line context (5 lines before and after)
- `watch(expr)` - Add expression to watch list
- `unwatch(expr)` - Remove expression from watch list

- `watchers` - List all watchers and their values (automatically listed on each breakpoint)
- `repl` - Open debugger's repl for evaluation in debugging script's context
- `exec expr` - Execute an expression in debugging script's context

Execution control

- `run` - Run script (automatically runs on debugger's start)
- `restart` - Restart script
- `kill` - Kill script

Various

- `scripts` - List all loaded scripts
- `version` - Display V8's version

Advanced Usage

V8 Inspector Integration for Node.js

V8 Inspector integration allows attaching Chrome DevTools to Node.js instances for debugging and profiling. It uses the Chrome Debugging Protocol.

V8 Inspector can be enabled by passing the `--inspect` flag when starting a Node.js application. It is also possible to supply a custom port with that flag, e.g. `--inspect=9222` will accept DevTools connections on port 9222.

To break on the first line of the application code, pass the `--inspect-brk` flag instead of `--inspect`.

```
$ node --inspect index.js
Debugger listening on 127.0.0.1:9229.
To start debugging, open the following URL in Chrome:
    chrome-devtools://devtools/bundled/inspector.html?experiments=true&v8only=true&ws=127.0.0.1:9229/dc9010dd-f8b8-4ac5-a510-c1a114ec7d29
```

(In the example above, the UUID dc9010dd-f8b8-4ac5-a510-c1a114ec7d29 at the end of the URL is generated on the fly, it varies in different debugging sessions.)

Deprecated APIs

Node.js may deprecate APIs when either: (a) use of the API is considered to be unsafe, (b) an improved alternative API has been made available, or (c) breaking changes to the API are expected in a future major release.

Node.js utilizes three kinds of Deprecations:

- Documentation-only
- Runtime
- End-of-Life

A Documentation-only deprecation is one that is expressed only within the Node.js API docs. These generate no side-effects while running Node.js. Some Documentation-only deprecations trigger a runtime warning when launched with [`--pending-deprecation`][] flag (or its alternative, `NODE_PENDING_DEPRECATION=1` environment variable), similarly to Runtime deprecations below. Documentation-only deprecations that support that flag are explicitly labeled as such in the list of Deprecated APIs.

A Runtime deprecation will, by default, generate a process warning that will be printed to `stderr` the first time the deprecated API is used. When the `--throw-deprecation` command-line flag is used, a Runtime deprecation will cause an error to be thrown.

An End-of-Life deprecation is used to identify code that either has been removed or will soon be removed from Node.js.

Un-deprecation

From time-to-time the deprecation of an API may be reversed. Such action may happen in either a semver-minor or semver-major release. In such situations, this document will be updated with information relevant to the decision. *However, the deprecation identifier will not be modified*.

List of Deprecated APIs

DEP0001: http.OutgoingMessage.prototype.flush

Type: Runtime

The `OutgoingMessage.prototype.flush()` method is deprecated. Use `OutgoingMessage.prototype.flushHeaders()` instead.

DEP0002: require('_linklist')

Type: End-of-Life

The `_linklist` module is deprecated. Please use a userland alternative.

DEP0003: _writableState.buffer

Type: Runtime

The `_writableState.buffer` property is deprecated. Use the `_writableState.getBuffer()` method instead.

DEP0004: CryptoStream.prototype.readyState

Type: Documentation-only

The `CryptoStream.prototype.readyState` property is deprecated and should not be used.

DEP0005: Buffer() constructor

Type: Documentation-only (supports [`--pending-deprecation`][])

The `Buffer()` function and `new Buffer()` constructor are deprecated due to API usability issues that can potentially lead to accidental security issues.

As an alternative, use of the following methods of constructing `Buffer` objects is strongly recommended:

- Buffer.alloc(size[, fill[, encoding]]) - Create a `Buffer` with *initialized* memory.
- Buffer.allocUnsafe(size) - Create a `Buffer` with *uninitialized* memory.
- Buffer.allocUnsafeSlow(size) - Create a `Buffer` with *uninitialized* memory.
- Buffer.from(array) - Create a `Buffer` with a copy of `array`
- Buffer.from(arrayBuffer[, byteOffset[, length]]) - Create a `Buffer` that wraps the given `arrayBuffer`.
- Buffer.from(buffer) - Create a `Buffer` that copies `buffer`.
- Buffer.from(string[, encoding]) - Create a `Buffer` that copies `string`.

DEP0006: child_process options.customFds

Type: Runtime

Within the child_process module's `spawn()`, `fork()`, and `exec()` methods, the `options.customFds` option is deprecated. The `options.stdio` option should be used instead.

DEP0007: Replace cluster worker.suicide with worker.exitedAfterDisconnect

Type: End-of-Life

In an earlier version of the Node.js `cluster`, a boolean property with the name `suicide` was added to the `Worker` object. The intent of this property was to provide an indication of how and why the `Worker` instance exited. In Node.js 6.0.0, the old property was deprecated and replaced with a new worker.exitedAfterDisconnect property. The old property name did not precisely describe the actual semantics and was unnecessarily emotion-laden.

DEP0008: require('constants')

Type: Documentation-only

The `constants` module has been deprecated. When requiring access to constants relevant to specific Node.js builtin modules, developers should instead refer to the `constants` property exposed by the relevant module. For instance, `require('fs').constants` and `require('os').constants`.

DEP0009: crypto.pbkdf2 without digest

Type: End-of-life

Use of the crypto.pbkdf2() API without specifying a digest was deprecated in Node.js 6.0 because the method defaulted to using the non-recommended `'SHA1'` digest. Previously, a deprecation warning was printed. Starting in Node.js 8.0.0, calling `crypto.pbkdf2()` or `crypto.pbkdf2Sync()` with an undefined `digest` will throw a `TypeError`.

DEP0010: crypto.createCredentials

Type: Runtime

The crypto.createCredentials() API is deprecated. Please use tls.createSecureContext() instead.

DEP0011: crypto.Credentials

Type: Runtime

The `crypto.Credentials` class is deprecated. Please use tls.SecureContext instead.

DEP0012: Domain.dispose

Type: End-of-Life

`Domain.dispose()` is removed. Recover from failed I/O actions explicitly via error event handlers set on the domain instead.

DEP0013: fs asynchronous function without callback

Type: Runtime

Calling an asynchronous function without a callback is deprecated.

DEP0014: fs.read legacy String interface

Type: End-of-Life

The fs.read() legacy String interface is deprecated. Use the Buffer API as mentioned in the documentation instead.

DEP0015: fs.readSync legacy String interface

Type: End-of-Life

The fs.readSync() legacy String interface is deprecated. Use the Buffer API as mentioned in the documentation instead.

DEP0016: GLOBAL/root

Type: Runtime

The `GLOBAL` and `root` aliases for the `global` property have been deprecated and should no longer be used.

DEP0017: Intl.v8BreakIterator

Type: End-of-Life

`Intl.v8BreakIterator` was a non-standard extension and has been removed. See Intl.Segmenter .

DEP0018: Unhandled promise rejections

Type: Runtime

Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

DEP0019: require('.') resolved outside directory

Type: Runtime

In certain cases, `require('.')` may resolve outside the package directory. This behavior is deprecated and will be removed in a future major Node.js release.

DEP0020: Server.connections

Type: Runtime

The Server.connections property is deprecated. Please use the Server.getConnections() method instead.

DEP0021: Server.listenFD

Type: Runtime

The `Server.listenFD()` method is deprecated. Please use `Server.listen({fd: <number>})` instead.

DEP0022: os.tmpDir()

Type: Runtime

The `os.tmpDir()` API is deprecated. Please use `os.tmpdir()` instead.

DEP0023: os.getNetworkInterfaces()

Type: Runtime

The `os.getNetworkInterfaces()` method is deprecated. Please use the `os.networkInterfaces` property instead.

DEP0024: REPLServer.prototype.convertToContext()

Type: End-of-Life

The `REPLServer.prototype.convertToContext()` API is deprecated and should not be used.

DEP0025: require('sys')

Type: Runtime

The `sys` module is deprecated. Please use the `util` module instead.

DEP0026: util.print()

Type: Runtime

The `util.print()` API is deprecated. Please use `console.log()` instead.

DEP0027: util.puts()

Type: Runtime

The `util.puts()` API is deprecated. Please use `console.log()` instead.

DEP0028: util.debug()

Type: Runtime

The `util.debug()` API is deprecated. Please use `console.error()` instead.

DEP0029: util.error()

Type: Runtime

The `util.error()` API is deprecated. Please use `console.error()` instead.

DEP0030: SlowBuffer

Type: Documentation-only

The `SlowBuffer` class has been deprecated. Please use `Buffer.allocUnsafeSlow(size)` instead.

DEP0031: ecdh.setPublicKey()

Type: Documentation-only

The `ecdh.setPublicKey()` method is now deprecated as its inclusion in the API is not useful.

DEP0032: domain module

Type: Documentation-only

The `domain` module is deprecated and should not be used.

DEP0033: EventEmitter.listenerCount()

Type: Documentation-only

The `EventEmitter.listenerCount(emitter, eventName)` API has been deprecated. Please use `emitter.listenerCount(eventName)` instead.

DEP0034: fs.exists(path, callback)

Type: Documentation-only

The `fs.exists(path, callback)` API has been deprecated. Please use `fs.stat()` or `fs.access()` instead.

DEP0035: fs.lchmod(path, mode, callback)

Type: Documentation-only

The `fs.lchmod(path, mode, callback)` API has been deprecated.

DEP0036: fs.lchmodSync(path, mode)

Type: Documentation-only

The `fs.lchmodSync(path, mode)` API has been deprecated.

DEP0037: fs.lchown(path, uid, gid, callback)

Type: Documentation-only

The fs.lchown(path, uid, gid, callback) API has been deprecated.

DEP0038: fs.lchownSync(path, uid, gid)

Type: Documentation-only

The fs.lchownSync(path, uid, gid) API has been deprecated.

DEP0039: require.extensions

Type: Documentation-only

The require.extensions property has been deprecated.

DEP0040: punycode module

Type: Documentation-only

The punycode module has been deprecated. Please use a userland alternative instead.

DEP0041: NODE_REPL_HISTORY_FILE environment variable

Type: Documentation-only

The `NODE_REPL_HISTORY_FILE` environment variable has been deprecated.

DEP0042: tls.CryptoStream

Type: Documentation-only

The tls.CryptoStream class has been deprecated. Please use tls.TLSSocket instead.

DEP0043: tls.SecurePair

Type: Documentation-only

The tls.SecurePair class has been deprecated. Please use tls.TLSSocket instead.

DEP0044: util.isArray()

Type: Documentation-only

The util.isArray() API has been deprecated. Please use `Array.isArray()` instead.

DEP0045: util.isBoolean()

Type: Documentation-only

The `util.isBoolean()` API has been deprecated.

DEP0046: util.isBuffer()

Type: Documentation-only

The `util.isBuffer()` API has been deprecated. Please use `Buffer.isBuffer()` instead.

DEP0047: util.isDate()

Type: Documentation-only

The `util.isDate()` API has been deprecated.

DEP0048: util.isError()

Type: Documentation-only

The `util.isError()` API has been deprecated.

DEP0049: util.isFunction()

Type: Documentation-only

The `util.isFunction()` API has been deprecated.

DEP0050: util.isNull()

Type: Documentation-only

The `util.isNull()` API has been deprecated.

DEP0051: util.isNullOrUndefined()

Type: Documentation-only

The `util.isNullOrUndefined()` API has been deprecated.

DEP0052: util.isNumber()

Type: Documentation-only

The `util.isNumber()` API has been deprecated.

DEP0053 util.isObject()

Type: Documentation-only

The `util.isObject()` API has been deprecated.

DEP0054: util.isPrimitive()

Type: Documentation-only

The `util.isPrimitive()` API has been deprecated.

DEP0055: util.isRegExp()

Type: Documentation-only

The `util.isRegExp()` API has been deprecated.

DEP0056: util.isString()

Type: Documentation-only

The `util.isString()` API has been deprecated.

DEP0057: util.isSymbol()

Type: Documentation-only

The `util.isSymbol()` API has been deprecated.

DEP0058: util.isUndefined()

Type: Documentation-only

The `util.isUndefined()` API has been deprecated.

DEP0059: util.log()

Type: Documentation-only

The `util.log()` API has been deprecated.

DEP0060: util._extend()

Type: Documentation-only

The `util._extend()` API has been deprecated.

DEP0061: fs.SyncWriteStream

Type: Runtime

The `fs.SyncWriteStream` class was never intended to be a publicly accessible API. No alternative API is available. Please use a userland alternative.

DEP0062: node --debug

Type: Runtime

`--debug` activates the legacy V8 debugger interface, which has been removed as of V8 5.8. It is replaced by Inspector which is activated with `--inspect` instead.

DEP0063: ServerResponse.prototype.writeHeader()

Type: Documentation-only

The `http` module `ServerResponse.prototype.writeHeader()` API has been deprecated. Please use `ServerResponse.prototype.writeHead()` instead.

The `ServerResponse.prototype.writeHeader()` method was never documented as an officially supported API.

DEP0064: tls.createSecurePair()

Type: Runtime

The `tls.createSecurePair()` API was deprecated in documentation in Node.js 0.11.3. Users should use `tls.Socket` instead.

DEP0065: repl.REPL_MODE_MAGIC and NODE_REPL_MODE=magic

Type: Documentation-only

The `repl` module's `REPL_MODE_MAGIC` constant, used for `replMode` option, has been deprecated. Its behavior has been functionally identical to that of `REPL_MODE_SLOPPY` since Node.js v6.0.0, when V8 5.0 was imported. Please use `REPL_MODE_SLOPPY` instead.

The `NODE_REPL_MODE` environment variable is used to set the underlying `replMode` of an interactive `node` session. Its default value, `magic`, is similarly deprecated in favor of `sloppy`.

DEP0066: outgoingMessage._headers, outgoingMessage._headerNames

Type: Documentation-only

The `http` module `outgoingMessage._headers` and `outgoingMessage._headerNames` properties have been deprecated. Please instead use one of the public methods (e.g. `outgoingMessage.getHeader()`, `outgoingMessage.getHeaders()`, `outgoingMessage.getHeaderNames()`, `outgoingMessage.hasHeader()`, `outgoingMessage.removeHeader()`, `outgoingMessage.setHeader()`) for working with outgoing headers.

The `outgoingMessage._headers` and `outgoingMessage._headerNames` properties were never documented as officially supported properties.

DEP0067: OutgoingMessage.prototype._renderHeaders

Type: Documentation-only

The `http` module `OutgoingMessage.prototype._renderHeaders()` API has been deprecated.

The `OutgoingMessage.prototype._renderHeaders` property was never documented as an officially supported API.

DEP0068: node debug

Type: Runtime

`node debug` corresponds to the legacy CLI debugger which has been replaced with a V8-inspector based CLI debugger available through `node inspect`.

DEP0069: vm.runInDebugContext(string)

Type: Runtime

The DebugContext will be removed in V8 soon and will not be available in Node 10+.

DebugContext was an experimental API.

DEP0070: async_hooks.currentId()

Type: End-of-Life

`async_hooks.currentId()` was renamed to `async_hooks.executionAsyncId()` for clarity.

This change was made while `async_hooks` was an experimental API.

DEP0071: async_hooks.triggerId()

Type: End-of-Life

`async_hooks.triggerId()` was renamed to `async_hooks.triggerAsyncId()` for clarity.

This change was made while `async_hooks` was an experimental API.

DEP0072: async_hooks.AsyncResource.triggerId()

Type: End-of-Life

`async_hooks.AsyncResource.triggerId()` was renamed to `async_hooks.AsyncResource.triggerAsyncId()` for clarity.

This change was made while `async_hooks` was an experimental API.

DEP0073: Several internal properties of net.Server

Type: Runtime

Accessing several internal, undocumented properties of `net.Server` instances with inappropriate names has been deprecated.

As the original API was undocumented and not generally useful for non-internal code, no replacement API is provided.

DEP0074: REPLServer.bufferedCommand

Type: Runtime

The `REPLServer.bufferedCommand` property was deprecated in favor of `REPLServer.clearBufferedCommand()` .

DEP0075: REPLServer.parseREPLKeyword()

Type: Runtime

`REPLServer.parseREPLKeyword()` was removed from userland visibility.

DEP0076: tls.parseCertString()

Type: Runtime

`tls.parseCertString()` is a trivial parsing helper that was made public by mistake. This function can usually be replaced with:

```
const querystring = require('querystring');
querystring.parse(str, '\n', '=');
```

This function is not completely equivalent to `querystring.parse()` . One difference is that `querystring.parse()` does url decoding:

```
> querystring.parse('%E5%A5%BD=1', '\n', '=');
{ '□': '1' }
> tls.parseCertString('%E5%A5%BD=1');
{ '%E5%A5%BD': '1' }
```

DEP0077: Module._debug()

Type: Runtime

`Module._debug()` has been deprecated.

The `Module._debug()` function was never documented as an officially supported API.

DEP0078: REPLServer.turnOffEditorMode()

Type: Runtime

`REPLServer.turnOffEditorMode()` was removed from userland visibility.

DEP0079: Custom inspection function on Objects via .inspect()

Type: Documentation-only

Using a property named `inspect` on an object to specify a custom inspection function for util.inspect() is deprecated. Use util.inspect.custom instead. For backwards compatibility with Node.js prior to version 6.4.0, both may be specified.

DEP0080: path._makeLong()

Type: Documentation-only

The internal `path._makeLong()` was not intended for public use. However, userland modules have found it useful. The internal API has been deprecated and replaced with an identical, public `path.toNamespacedPath()` method.

DEP0081: fs.truncate() using a file descriptor

Type: Runtime

`fs.truncate()` `fs.truncateSync()` usage with a file descriptor has been deprecated. Please use `fs.ftruncate()` or `fs.ftruncateSync()` to work with file descriptors.

DEP0082: REPLServer.prototype.memory()

Type: Runtime

`REPLServer.prototype.memory()` is a function only necessary for the internal mechanics of the `REPLServer` itself, and is therefore not necessary in user space.

DEP0083: Disabling ECDH by setting ecdhCurve to false

Type: Runtime

The `ecdhCurve` option to `tls.createSecureContext()` and `tls.TLSSocket` could be set to `false` to disable ECDH entirely on the server only. This mode is deprecated in preparation for migrating to OpenSSL 1.1.0 and consistency with the client. Use the `ciphers` parameter instead.

DEP0085: AsyncHooks Sensitive API

Type: Runtime

The AsyncHooks Sensitive API was never documented and had various of minor issues, see https://github.com/nodejs/node/issues/15572 . Use the `AsyncResource` API instead.

DEP0086: Remove runInAsyncIdScope

Type: Runtime

`runInAsyncIdScope` doesn't emit the `before` or `after` event and can thus cause a lot of issues. See https://github.com/nodejs/node/issues/14328 for more details.

DEP0089: require('assert')

Type: Documentation-only

Importing assert directly is not recommended as the exposed functions will use loose equality checks. Use `require('assert').strict` instead. The API is the same as the legacy assert but it will always use strict equality checks.

DEP0098: AsyncHooks Embedder AsyncResource.emit<Before,After> APIs

Type: Runtime

The embedded API provided by AsyncHooks exposes emit<Before,After> methods which are very easy to use incorrectly which can lead to unrecoverable errors.

Use `asyncResource.runInAsyncScope()` API instead which provides a much safer, and more convenient, alternative. See https://github.com/nodejs/node/pull/18513 for more details.

DNS

Stability: 2 - Stable

The `dns` module contains functions belonging to two different categories:

1) Functions that use the underlying operating system facilities to perform name resolution, and that do not necessarily perform any network communication. This category contains only one function: `dns.lookup()`. **Developers looking to perform name resolution in the same way that other applications on the same operating system behave should use `dns.lookup()`.**

For example, looking up `iana.org`.

```
const dns = require('dns');

dns.lookup('iana.org', (err, address, family) => {
  console.log('address: %j family: IPv%s', address, family);
});
// address: "192.0.43.8" family: IPv4
```

2) Functions that connect to an actual DNS server to perform name resolution, and that *always* use the network to perform DNS queries. This category contains all functions in the `dns` module *except* `dns.lookup()`. These functions do not use the same set of configuration files used by `dns.lookup()` (e.g. `/etc/hosts`). These functions should be used by developers who do not want to use the underlying operating system's facilities for name resolution, and instead want to *always* perform DNS queries.

Below is an example that resolves `'archive.org'` then reverse resolves the IP addresses that are returned.

```
const dns = require('dns');

dns.resolve4('archive.org', (err, addresses) => {
  if (err) throw err;

  console.log(`addresses: ${JSON.stringify(addresses)}`);

  addresses.forEach((a) => {
    dns.reverse(a, (err, hostnames) => {
      if (err) {
        throw err;
      }
      console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
    });
  });
});
```

There are subtle consequences in choosing one over the other, please consult the Implementation considerations section for more information.

Class dns.Resolver

Added in: v8.3.0

An independent resolver for DNS requests.

Note that creating a new resolver uses the default server settings. Setting the servers used for a resolver using `resolver.setServers()` does not affect other resolver:

```
const { Resolver } = require('dns');
const resolver = new Resolver();
resolver.setServers(['4.4.4.4']);

// This request will use the server at 4.4.4.4, independent of global settings.
resolver.resolve4('example.org', (err, addresses) => {
  // ...
});
```

The following methods from the `dns` module are available:

- resolver.getServers()
- resolver.setServers()
- resolver.resolve()
- resolver.resolve4()
- resolver.resolve6()
- resolver.resolveAny()
- resolver.resolveCname()
- resolver.resolveMx()
- resolver.resolveNaptr()
- resolver.resolveNs()
- resolver.resolvePtr()
- resolver.resolveSoa()
- resolver.resolveSrv()
- resolver.resolveTxt()

- resolver.reverse()

resolver.cancel()

Added in: v8.3.0

Cancel all outstanding DNS queries made by this resolver. The corresponding callbacks will be called with an error with code `ECANCELLED`.

dns.getServers()

Added in: v0.11.3

Returns an array of IP address strings, formatted according to rfc5952, that are currently configured for DNS resolution. A string will include a port section if a custom port is used.

```
[
  '4.4.4.4',
  '2001:4860:4860::8888',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053'
]
```

dns.lookup(hostname[, options], callback)

▸ History

- `hostname` <string>
- `options` <integer> | <Object>
  - `family` <integer> The record family. Must be `4` or `6`. IPv4 and IPv6 addresses are both returned by default.
  - `hints` <number> One or more supported getaddrinfo flags. Multiple flags may be passed by bitwise `OR`ing their values.
  - `all` <boolean> When `true`, the callback returns all resolved addresses in an array. Otherwise, returns a single address. **Default:** `false`
  - `verbatim` <boolean> When `true`, the callback receives IPv4 and IPv6 addresses in the order the DNS resolver returned them. When `false`, IPv4 addresses are placed before IPv6 addresses. **Default:** currently `false` (addresses are reordered) but this is expected to change in the not too distant future. New code should use `{ verbatim: true }`.
- `callback` <Function>
  - `err` <Error>
  - `address` <string> A string representation of an IPv4 or IPv6 address.
  - `family` <integer> `4` or `6`, denoting the family of `address`.

Resolves a hostname (e.g. `'nodejs.org'`) into the first found A (IPv4) or AAAA (IPv6) record. All `option` properties are optional. If `options` is an integer, then it must be `4` or `6` – if `options` is not provided, then IPv4 and IPv6 addresses are both returned if found.

With the `all` option set to `true`, the arguments for `callback` change to `(err, addresses)`, with `addresses` being an array of objects with the properties `address` and `family`.

On error, `err` is an Error object, where `err.code` is the error code. Keep in mind that `err.code` will be set to `'ENOENT'` not only when the hostname does not exist but also when the lookup fails in other ways such as no available file descriptors.

`dns.lookup()` does not necessarily have anything to do with the DNS protocol. The implementation uses an operating system facility that can associate names with addresses, and vice versa. This implementation can have subtle but important consequences on the behavior of any Node.js program. Please take some time to consult the Implementation considerations section before using `dns.lookup()`.

Example usage:

```
const dns = require('dns');
const options = {
  family: 6,
  hints: dns.ADDRCONFIG | dns.V4MAPPED,
};
dns.lookup('example.com', options, (err, address, family) =>
  console.log('address: %j family: IPv%s', address, family));
// address: "2606:2800:220:1:248:1893:25c8:1946" family: IPv6

// When options.all is true, the result will be an Array.
options.all = true;
dns.lookup('example.com', options, (err, addresses) =>
  console.log('addresses: %j', addresses));
// addresses: [{"address":"2606:2800:220:1:248:1893:25c8:1946","family":6}]
```

If this method is invoked as its `util.promisify()` ed version, and `all` is not set to `true`, it returns a Promise for an object with `address` and `family` properties.

Supported getaddrinfo flags

The following flags can be passed as hints to `dns.lookup()` .

- `dns.ADDRCONFIG` : Returned address types are determined by the types of addresses supported by the current system. For example, IPv4 addresses are only returned if the current system has at least one IPv4 address configured. Loopback addresses are not considered.
- `dns.V4MAPPED` : If the IPv6 family was specified, but no IPv6 addresses were found, then return IPv4 mapped IPv6 addresses. Note that it is not supported on some operating systems (e.g FreeBSD 10.1).

dns.lookupService(address, port, callback)

Added in: v0.11.14

- `address` <string>
- `port` <number>
- `callback` <Function>
  - `err` <Error>
  - `hostname` <string> e.g. `example.com`
  - `service` <string> e.g. `http`

Resolves the given `address` and `port` into a hostname and service using the operating system's underlying `getnameinfo` implementation.

If `address` is not a valid IP address, a `TypeError` will be thrown. The `port` will be coerced to a number. If it is not a legal port, a `TypeError` will be thrown.

On an error, `err` is an `Error` object, where `err.code` is the error code.

```
const dns = require('dns');
dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {
  console.log(hostname, service);
  // Prints: localhost ssh
});
```

If this method is invoked as its `util.promisify()` ed version, it returns a Promise for an object with `hostname` and `service` properties.

dns.resolve(hostname[, rrtype], callback)

Added in: v0.1.27

- `hostname` <string> Hostname to resolve.

- rrtype <string> Resource record type. **Default:** 'A'
- callback <Function>
  - err <Error>
  - records <string[]> | <Object[]> | <Object>

Uses the DNS protocol to resolve a hostname (e.g. 'nodejs.org') into an array of the resource records. The callback function has arguments (err, records) . When successful, records will be an array of resource records. The type and structure of individual results varies based on rrtype :

rrtype	records contains	Result type	Shorthand method
'A'	IPv4 addresses (default)	<string>	dns.resolve4()
'AAAA'	IPv6 addresses	<string>	dns.resolve6()
'CNAME'	canonical name records	<string>	dns.resolveCname()
'MX'	mail exchange records	<Object>	dns.resolveMx()
'NAPTR'	name authority pointer records	<Object>	dns.resolveNaptr()
'NS'	name server records	<string>	dns.resolveNs()
'PTR'	pointer records	<string>	dns.resolvePtr()
'SOA'	start of authority records	<Object>	dns.resolveSoa()
'SRV'	service records	<Object>	dns.resolveSrv()
'TXT'	text records	<string[]>	dns.resolveTxt()
'ANY'	any records	<Object>	dns.resolveAny()

On error, err is an Error object, where err.code is one of the DNS error codes .

dns.resolve4(hostname[, options], callback)

▸ History

- hostname <string> Hostname to resolve.
- options <Object>
  - ttl <boolean> Retrieve the Time-To-Live value (TTL) of each record. When true , the callback receives an array of { address: '1.2.3.4', ttl: 60 } objects rather than an array of strings, with the TTL expressed in seconds.
- callback <Function>
  - err <Error>
  - addresses <string[]> | <Object[]>

Uses the DNS protocol to resolve a IPv4 addresses (A records) for the hostname . The addresses argument passed to the callback function will contain an array of IPv4 addresses (e.g. ['74.125.79.104', '74.125.79.105', '74.125.79.106']).

dns.resolve6(hostname[, options], callback)

▸ History

- hostname <string> Hostname to resolve.
- options <Object>

- `ttl` <boolean> Retrieve the Time-To-Live value (TTL) of each record. When `true`, the callback receives an array of `{ address: '0:1:2:3:4:5:6:7', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.

- `callback` <Function>
  - `err` <Error>
  - `addresses` <string[]> | <Object[]>

Uses the DNS protocol to resolve a IPv6 addresses (`AAAA` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of IPv6 addresses.

dns.resolveCname(hostname, callback)

Added in: v0.3.2

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `addresses` <string[]>

Uses the DNS protocol to resolve `CNAME` records for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of canonical name records available for the `hostname` (e.g. `['bar.example.com']`).

dns.resolveMx(hostname, callback)

Added in: v0.1.27

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `addresses` <Object[]>

Uses the DNS protocol to resolve mail exchange records (`MX` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of objects containing both a `priority` and `exchange` property (e.g. `[{priority: 10, exchange: 'mx.example.com'}, ...]`).

dns.resolveNaptr(hostname, callback)

Added in: v0.9.12

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `addresses` <Object[]>

Uses the DNS protocol to resolve regular expression based records (`NAPTR` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of objects with the following properties:

- `flags`
- `service`
- `regexp`
- `replacement`
- `order`
- `preference`

```
{
  flags: 's',
  service: 'SIP+D2U',
  regexp: '',
  replacement: '_sip._udp.example.com',
  order: 30,
  preference: 100
}
```

dns.resolveNs(hostname, callback)

Added in: v0.1.90

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `addresses` <string[]>

Uses the DNS protocol to resolve name server records (`NS` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of name server records available for `hostname` (e.g. `['ns1.example.com', 'ns2.example.com']`).

dns.resolvePtr(hostname, callback)

Added in: v6.0.0

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `addresses` <string[]>

Uses the DNS protocol to resolve pointer records (`PTR` records) for the `hostname`. The `addresses` argument passed to the `callback` function will be an array of strings containing the reply records.

dns.resolveSoa(hostname, callback)

Added in: v0.11.10

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `address` <Object>

Uses the DNS protocol to resolve a start of authority record (`SOA` record) for the `hostname`. The `address` argument passed to the `callback` function will be an object with the following properties:

- `nsname`
- `hostmaster`
- `serial`
- `refresh`
- `retry`
- `expire`
- `minttl`

```
{
  nsname: 'ns.example.com',
  hostmaster: 'root.example.com',
  serial: 2013101809,
  refresh: 10000,
  retry: 2400,
  expire: 604800,
  minttl: 3600
}
```

dns.resolveSrv(hostname, callback)

Added in: v0.1.27

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `addresses` <Object[]>

Uses the DNS protocol to resolve service records (`SRV` records) for the `hostname`. The `addresses` argument passed to the `callback` function will be an array of objects with the following properties:

- `priority`
- `weight`
- `port`
- `name`

```
{
  priority: 10,
  weight: 5,
  port: 21223,
  name: 'service.example.com'
}
```

dns.resolveTxt(hostname, callback)

Added in: v0.1.27

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `records` <string[][]>

Uses the DNS protocol to resolve text queries (`TXT` records) for the `hostname`. The `records` argument passed to the `callback` function is a two-dimensional array of the text records available for `hostname` (e.g. `[ ['v=spf1 ip4:0.0.0.0 ', '~all' ] ]`). Each sub-array contains TXT chunks of one record. Depending on the use case, these could be either joined together or treated separately.

dns.resolveAny(hostname, callback)

- `hostname` <string>
- `callback` <Function>
  - `err` <Error>
  - `ret` <Object[]>

Uses the DNS protocol to resolve all records (also known as `ANY` or `*` query). The `ret` argument passed to the `callback` function will be an array containing various types of records. Each object has a property `type` that indicates the type of the current record. And depending on the

type , additional properties will be present on the object:

Type	Properties
'A'	address / ttl
'AAAA'	address / ttl
'CNAME'	value
'MX'	Refer to dns.resolveMx()
'NAPTR'	Refer to dns.resolveNaptr()
'NS'	value
'PTR'	value
'SOA'	Refer to dns.resolveSoa()
'SRV'	Refer to dns.resolveSrv()
'TXT'	This type of record contains an array property called entries which refers to dns.resolveTxt() , eg. { entries: ['...'], type: 'TXT' }

Here is an example of the ret object passed to the callback:

```
[ { type: 'A', address: '127.0.0.1', ttl: 299 },
  { type: 'CNAME', value: 'example.com' },
  { type: 'MX', exchange: 'alt4.aspmx.l.example.com', priority: 50 },
  { type: 'NS', value: 'ns1.example.com' },
  { type: 'TXT', entries: [ 'v=spf1 include:_spf.example.com ~all' ] },
  { type: 'SOA',
    nsname: 'ns1.example.com',
    hostmaster: 'admin.example.com',
    serial: 156696742,
    refresh: 900,
    retry: 900,
    expire: 1800,
    minttl: 60 } ]
```

dns.reverse(ip, callback)

Added in: v0.1.16

- ip <string>
- callback <Function>
  - err <Error>
  - hostnames <string[]>

Performs a reverse DNS query that resolves an IPv4 or IPv6 address to an array of hostnames.

On error, err is an Error object, where err.code is one of the DNS error codes .

dns.setServers(servers)

Added in: v0.11.3

- servers <string[]> array of rfc5952 formatted addresses

Sets the IP address and port of servers to be used when performing DNS resolution. The `servers` argument is an array of rfc5952 formatted addresses. If the port is the IANA default DNS port (53) it can be omitted.

```
dns.setServers([
  '4.4.4.4',
  '[2001:4860:4860::8888]',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053'
]);
```

An error will be thrown if an invalid address is provided.

The `dns.setServers()` method must not be called while a DNS query is in progress.

Error codes

Each DNS query can return one of the following error codes:

- `dns.NODATA`: DNS server returned answer with no data.
- `dns.FORMERR`: DNS server claims query was misformatted.
- `dns.SERVFAIL`: DNS server returned general failure.
- `dns.NOTFOUND`: Domain name not found.
- `dns.NOTIMP`: DNS server does not implement requested operation.
- `dns.REFUSED`: DNS server refused query.
- `dns.BADQUERY`: Misformatted DNS query.
- `dns.BADNAME`: Misformatted hostname.
- `dns.BADFAMILY`: Unsupported address family.
- `dns.BADRESP`: Misformatted DNS reply.
- `dns.CONNREFUSED`: Could not contact DNS servers.
- `dns.TIMEOUT`: Timeout while contacting DNS servers.
- `dns.EOF`: End of file.
- `dns.FILE`: Error reading file.
- `dns.NOMEM`: Out of memory.
- `dns.DESTRUCTION`: Channel is being destroyed.
- `dns.BADSTR`: Misformatted string.
- `dns.BADFLAGS`: Illegal flags specified.
- `dns.NONAME`: Given hostname is not numeric.
- `dns.BADHINTS`: Illegal hints flags specified.
- `dns.NOTINITIALIZED`: c-ares library initialization not yet performed.
- `dns.LOADIPHLPAPI`: Error loading iphlpapi.dll.
- `dns.ADDRGETNETWORKPARAMS`: Could not find GetNetworkParams function.
- `dns.CANCELLED`: DNS query cancelled.

Implementation considerations

Although `dns.lookup()` and the various `dns.resolve*()/dns.reverse()` functions have the same goal of associating a network name with a network address (or vice versa), their behavior is quite different. These differences can have subtle but significant consequences on the behavior of Node.js programs.

`dns.lookup()`

Under the hood, `dns.lookup()` uses the same operating system facilities as most other programs. For instance, `dns.lookup()` will almost always resolve a given name the same way as the `ping` command. On most POSIX-like operating systems, the behavior of the `dns.lookup()` function can be modified by changing settings in nsswitch.conf(5) and/or resolv.conf(5), but note that changing these files will change the behavior of *all other programs running on the same operating system.*

Though the call to `dns.lookup()` will be asynchronous from JavaScript's perspective, it is implemented as a synchronous call to getaddrinfo(3) that runs on libuv's threadpool. This can have surprising negative performance implications for some applications, see the `UV_THREADPOOL_SIZE` documentation for more information.

Note that various networking APIs will call `dns.lookup()` internally to resolve host names. If that is an issue, consider resolving the hostname to and address using `dns.resolve()` and using the address instead of a host name. Also, some networking APIs (such as `socket.connect()` and `dgram.createSocket()`) allow the default resolver, `dns.lookup()`, to be replaced.

`dns.resolve()`, `dns.resolve*()` and `dns.reverse()`

These functions are implemented quite differently than `dns.lookup()`. They do not use getaddrinfo(3) and they *always* perform a DNS query on the network. This network communication is always done asynchronously, and does not use libuv's threadpool.

As a result, these functions cannot have the same negative impact on other processing that happens on libuv's threadpool that `dns.lookup()` can have.

They do not use the same set of configuration files than what `dns.lookup()` uses. For instance, *they do not use the configuration from* `/etc/hosts`.

Domain

▸ History

Stability: 0 - Deprecated

This module is pending deprecation. Once a replacement API has been finalized, this module will be fully deprecated. Most end users should **not** have cause to use this module. Users who absolutely must have the functionality that domains provide may rely on it for the time being but should expect to have to migrate to a different solution in the future.

Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an `'error'` event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the `process.on('uncaughtException')` handler, or causing the program to exit immediately with an error code.

Warning: Don't Ignore Errors!

Domain error handlers are not a substitute for closing down a process when an error occurs.

By the very nature of how `throw` works in JavaScript, there is almost never any way to safely "pick up where you left off", without leaking references, or creating some other sort of undefined brittle state.

The safest way to respond to a thrown error is to shut down the process. Of course, in a normal web server, there may be many open connections, and it is not reasonable to abruptly shut those down because an error was triggered by someone else.

The better approach is to send an error response to the request that triggered the error, while letting the others finish in their normal time, and stop listening for new requests in that worker.

In this way, `domain` usage goes hand-in-hand with the cluster module, since the master process can fork a new worker when a worker encounters an error. For Node.js programs that scale to multiple machines, the terminating proxy or service registry can take note of the failure, and react accordingly.

For example, this is not a good idea:

```
// XXX WARNING! BAD IDEA!

const d = require('domain').create();
d.on('error', (er) => {
  // The error won't crash the process, but what it does is worse!
  // Though we've prevented abrupt process restarting, we are leaking
  // resources like crazy if this ever happens.
  // This is no better than process.on('uncaughtException')!
  console.log(`error, but oh well ${er.message}`);
});
d.run(() => {
  require('http').createServer((req, res) => {
    handleRequest(req, res);
  }).listen(PORT);
});
```

By using the context of a domain, and the resilience of separating our program into multiple worker processes, we can react more appropriately, and handle errors with much greater safety.

```
// Much better!

const cluster = require('cluster');
const PORT = +process.env.PORT || 1337;

if (cluster.isMaster) {
  // A more realistic scenario would have more than 2 workers,
  // and perhaps not put the master and worker in the same file.
  //
  // It is also possible to get a bit fancier about logging, and
  // implement whatever custom logic is needed to prevent DoS
  // attacks and other bad behavior.
  //
  // See the options in the cluster documentation.
  //
  // The important thing is that the master does very little,
  // increasing our resilience to unexpected errors.

  cluster.fork();
  cluster.fork();

  cluster.on('disconnect', (worker) => {
    console.error('disconnect!');
    cluster.fork();
  });

} else {
  // the worker
  //
  // This is where we put our bugs!

  const domain = require('domain');

  // See the cluster documentation for more details about using
  // worker processes to serve requests. How it works, caveats, etc.

  const server = require('http').createServer((req, res) => {
    const d = domain.create();
```

```
    d.on('error', (er) => {
      console.error(`error ${er.stack}`);

      // Note: We're in dangerous territory!
      // By definition, something unexpected occurred,
      // which we probably didn't want.
      // Anything can happen now! Be very careful!

      try {
        // make sure we close down within 30 seconds
        const killtimer = setTimeout(() => {
          process.exit(1);
        }, 30000);
        // But don't keep the process open just for that!
        killtimer.unref();

        // stop taking new requests.
        server.close();

        // Let the master know we're dead. This will trigger a
        // 'disconnect' in the cluster master, and then it will fork
        // a new worker.
        cluster.worker.disconnect();

        // try to send an error to the request that triggered the problem
        res.statusCode = 500;
        res.setHeader('content-type', 'text/plain');
        res.end('Oops, there was a problem!\n');
      } catch (er2) {
        // oh well, not much we can do at this point.
        console.error(`Error sending 500! ${er2.stack}`);
      }
    });

    // Because req and res were created before this domain existed,
    // we need to explicitly add them.
    // See the explanation of implicit vs explicit binding below.
    d.add(req);
    d.add(res);

    // Now run the handler function in the domain.
    d.run(() => {
      handleRequest(req, res);
    });
  });
  server.listen(PORT);
}

// This part is not important. Just an example routing thing.
// Put fancy application logic here.
function handleRequest(req, res) {
  switch (req.url) {
    case '/error':
      // We do some async stuff, and then...
      setTimeout(() => {
        // Whoops!
        flerb.bark();
      }, timeout);
```

```
      }, timeout);
      break;
    default:
      res.end('ok');
  }
}
```

Additions to Error objects

Any time an `Error` object is routed through a domain, a few extra fields are added to it.

- `error.domain` The domain that first handled the error.
- `error.domainEmitter` The event emitter that emitted an `'error'` event with the error object.
- `error.domainBound` The callback function which was bound to the domain, and passed an error as its first argument.
- `error.domainThrown` A boolean indicating whether the error was thrown, emitted, or passed to a bound callback function.

Implicit Binding

If domains are in use, then all **new** EventEmitter objects (including Stream objects, requests, responses, etc.) will be implicitly bound to the active domain at the time of their creation.

Additionally, callbacks passed to lowlevel event loop requests (such as to fs.open, or other callback-taking methods) will automatically be bound to the active domain. If they throw, then the domain will catch the error.

In order to prevent excessive memory usage, Domain objects themselves are not implicitly added as children of the active domain. If they were, then it would be too easy to prevent request and response objects from being properly garbage collected.

To nest Domain objects as children of a parent Domain they must be explicitly added.

Implicit binding routes thrown errors and `'error'` events to the Domain's `'error'` event, but does not register the EventEmitter on the Domain. Implicit binding only takes care of thrown errors and `'error'` events.

Explicit Binding

Sometimes, the domain in use is not the one that ought to be used for a specific event emitter. Or, the event emitter could have been created in the context of one domain, but ought to instead be bound to some other domain.

For example, there could be one domain in use for an HTTP server, but perhaps we would like to have a separate domain to use for each request.

That is possible via explicit binding.

```
// create a top-level domain for the server
const domain = require('domain');
const http = require('http');
const serverDomain = domain.create();

serverDomain.run(() => {
  // server is created in the scope of serverDomain
  http.createServer((req, res) => {
    // req and res are also created in the scope of serverDomain
    // however, we'd prefer to have a separate domain for each request.
    // create it first thing, and add req and res to it.
    const reqd = domain.create();
    reqd.add(req);
    reqd.add(res);
    reqd.on('error', (er) => {
      console.error('Error', er, req.url);
      try {
        res.writeHead(500);
        res.end('Error occurred, sorry.');
      } catch (er2) {
        console.error('Error sending 500', er2, req.url);
      }
    });
  }).listen(1337);
});
```

domain.create()

- Returns: <Domain>

Returns a new Domain object.

Class: Domain

The Domain class encapsulates the functionality of routing errors and uncaught exceptions to the active Domain object.

Domain is a child class of EventEmitter . To handle the errors that it catches, listen to its `'error'` event.

domain.members

- <Array>

An array of timers and event emitters that have been explicitly added to the domain.

domain.add(emitter)

- `emitter` <EventEmitter> | <Timer> emitter or timer to be added to the domain

Explicitly adds an emitter to the domain. If any event handlers called by the emitter throw an error, or if the emitter emits an `'error'` event, it will be routed to the domain's `'error'` event, just like with implicit binding.

This also works with timers that are returned from setInterval() and setTimeout() . If their callback function throws, it will be caught by the domain 'error' handler.

If the Timer or EventEmitter was already bound to a domain, it is removed from that one, and bound to this one instead.

domain.bind(callback)

- `callback` <Function> The callback function

- Returns: <Function> The bound function

The returned function will be a wrapper around the supplied callback function. When the returned function is called, any errors that are thrown will be routed to the domain's `'error'` event.

Example

```
const d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.bind((er, data) => {
    // if this throws, it will also be passed to the domain
    return cb(er, data ? JSON.parse(data) : null);
  }));
}

d.on('error', (er) => {
  // an error occurred somewhere.
  // if we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

domain.enter()

The `enter` method is plumbing used by the `run`, `bind`, and `intercept` methods to set the active domain. It sets `domain.active` and `process.domain` to the domain, and implicitly pushes the domain onto the domain stack managed by the domain module (see domain.exit() for details on the domain stack). The call to `enter` delimits the beginning of a chain of asynchronous calls and I/O operations bound to a domain.

Calling `enter` changes only the active domain, and does not alter the domain itself. `enter` and `exit` can be called an arbitrary number of times on a single domain.

domain.exit()

The `exit` method exits the current domain, popping it off the domain stack. Any time execution is going to switch to the context of a different chain of asynchronous calls, it's important to ensure that the current domain is exited. The call to `exit` delimits either the end of or an interruption to the chain of asynchronous calls and I/O operations bound to a domain.

If there are multiple, nested domains bound to the current execution context, `exit` will exit any domains nested within this domain.

Calling `exit` changes only the active domain, and does not alter the domain itself. `enter` and `exit` can be called an arbitrary number of times on a single domain.

domain.intercept(callback)

- `callback` <Function> The callback function
- Returns: <Function> The intercepted function

This method is almost identical to domain.bind(callback). However, in addition to catching thrown errors, it will also intercept Error objects sent as the first argument to the function.

In this way, the common `if (err) return callback(err);` pattern can be replaced with a single error handler in a single place.

Example

```
const d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.intercept((data) => {
    // note, the first argument is never passed to the
    // callback since it is assumed to be the 'Error' argument
    // and thus intercepted by the domain.

    // if this throws, it will also be passed to the domain
    // so the error-handling logic can be moved to the 'error'
    // event on the domain instead of being repeated throughout
    // the program.
    return cb(null, JSON.parse(data));
  }));
}

d.on('error', (er) => {
  // an error occurred somewhere.
  // if we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

domain.remove(emitter)

- `emitter` <EventEmitter> | <Timer> emitter or timer to be removed from the domain

The opposite of `domain.add(emitter)`. Removes domain handling from the specified emitter.

domain.run(fn[, ...args])

- `fn` <Function>
- `...args` <any>

Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and lowlevel requests that are created in that context. Optionally, arguments can be passed to the function.

This is the most basic way to use a domain.

Example:

```
const domain = require('domain');
const fs = require('fs');
const d = domain.create();
d.on('error', (er) => {
  console.error('Caught error!', er);
});
d.run(() => {
  process.nextTick(() => {
    setTimeout(() => { // simulating some various async stuff
      fs.open('non-existent file', 'r', (er, fd) => {
        if (er) throw er;
        // proceed...
      });
    }, 100);
  });
});
```

In this example, the `d.on('error')` handler will be triggered, rather than crashing the program.

Domains and Promises

As of Node 8.0.0, the handlers of Promises are run inside the domain in which the call to `.then` or `.catch` itself was made:

```
const d1 = domain.create();
const d2 = domain.create();

let p;
d1.run(() => {
  p = Promise.resolve(42);
});

d2.run(() => {
  p.then((v) => {
    // running in d2
  });
});
```

A callback may be bound to a specific domain using domain.bind(callback) :

```
const d1 = domain.create();
const d2 = domain.create();

let p;
d1.run(() => {
  p = Promise.resolve(42);
});

d2.run(() => {
  p.then(p.domain.bind((v) => {
    // running in d1
  }));
});
```

Note that domains will not interfere with the error handling mechanisms for Promises, i.e. no `error` event will be emitted for unhandled Promise rejections.

ECMAScript Modules

Stability: 1 - Experimental

Node.js contains support for ES Modules based upon the Node.js EP for ES Modules .

Not all features of the EP are complete and will be landing as both VM support and implementation is ready. Error messages are still being polished.

Enabling

The `--experimental-modules` flag can be used to enable features for loading ESM modules.

Once this has been set, files ending with `.mjs` will be able to be loaded as ES Modules.

```
node --experimental-modules my-app.mjs
```

Features

Supported

Only the CLI argument for the main entry point to the program can be an entry point into an ESM graph. Dynamic import can also be used to create entry points into ESM graphs at runtime.

Unsupported

Feature	Reason
`require('./foo.mjs')`	ES Modules have differing resolution and timing, use language standard `import()`
`import()`	pending newer V8 release used in Node.js
`import.meta`	pending V8 implementation

Notable differences between `import` and `require`

No NODE_PATH

`NODE_PATH` is not part of resolving `import` specifiers. Please use symlinks if this behavior is desired.

No `require.extensions`

`require.extensions` is not used by `import`. The expectation is that loader hooks can provide this workflow in the future.

No `require.cache`

`require.cache` is not used by `import`. It has a separate cache.

URL based paths

ESM are resolved and cached based upon URL semantics. This means that files containing special characters such as `#` and `?` need to be escaped.

Modules will be loaded multiple times if the `import` specifier used to resolve them have a different query or fragment.

```
import './foo?query=1'; // loads ./foo with query of "?query=1"
import './foo?query=2'; // loads ./foo with query of "?query=2"
```

For now, only modules using the `file:` protocol can be loaded.

Interop with existing modules

All CommonJS, JSON, and C++ modules can be used with `import`.

Modules loaded this way will only be loaded once, even if their query or fragment string differs between `import` statements.

When loaded via `import` these modules will provide a single `default` export representing the value of `module.exports` at the time they finished evaluating.

```
import fs from 'fs';
fs.readFile('./foo.txt', (err, body) => {
  if (err) {
    console.error(err);
  } else {
    console.log(body);
  }
});
```

Loader hooks

To customize the default module resolution, loader hooks can optionally be provided via a `--loader ./loader-name.mjs` argument to Node.

When hooks are used they only apply to ES module loading and not to any CommonJS modules loaded.

Resolve hook

The resolve hook returns the resolved file URL and module format for a given module specifier and parent file URL:

```
const baseURL = new URL('file://');
baseURL.pathname = `${process.cwd()}/`;

export async function resolve(specifier,
                              parentModuleURL = baseURL,
                              defaultResolver) {
  return {
    url: new URL(specifier, parentModuleURL).href,
    format: 'esm'
  };
}
```

The parentURL is provided as `undefined` when performing main Node.js load itself.

The default Node.js ES module resolution function is provided as a third argument to the resolver for easy compatibility workflows.

In addition to returning the resolved file URL value, the resolve hook also returns a `format` property specifying the module format of the resolved module. This can be one of the following:

`format`	Description
`'esm'`	Load a standard JavaScript module
`'cjs'`	Load a node-style CommonJS module
`'builtin'`	Load a node builtin CommonJS module
`'json'`	Load a JSON file
`'addon'`	Load a C++ Addon
`'dynamic'`	Use a dynamic instantiate hook

For example, a dummy loader to load JavaScript restricted to browser resolution rules with only JS file extension and Node builtin modules support could be written:

```
import url from 'url';
import path from 'path';
import process from 'process';
import Module from 'module';

const builtins = Module.builtinModules;
const JS_EXTENSIONS = new Set(['.js', '.mjs']);

const baseURL = new URL('file://');
baseURL.pathname = `${process.cwd()}/`;

export function resolve(specifier, parentModuleURL = baseURL, defaultResolve) {
  if (builtins.includes(specifier)) {
    return {
      url: specifier,
      format: 'builtin'
    };
  }
  if (/^\.{0,2}[/]/.test(specifier) !== true && !specifier.startsWith('file:')) {
    // For node_modules support:
    // return defaultResolve(specifier, parentModuleURL);
    throw new Error(
      `imports must begin with '/', './', or '../'; '${specifier}' does not`);
  }
  const resolved = new url.URL(specifier, parentModuleURL);
  const ext = path.extname(resolved.pathname);
  if (!JS_EXTENSIONS.has(ext)) {
    throw new Error(
      `Cannot load file with non-JavaScript file extension ${ext}.`);
  }
  return {
    url: resolved.href,
    format: 'esm'
  };
}
```

With this loader, running:

```
NODE_OPTIONS='--experimental-modules --loader ./custom-loader.mjs' node x.js
```

would load the module `x.js` as an ES module with relative resolution support (with `node_modules` loading skipped in this example).

Dynamic instantiate hook

To create a custom dynamic module that doesn't correspond to one of the existing `format` interpretations, the `dynamicInstantiate` hook can be used. This hook is called only for modules that return `format: 'dynamic'` from the `resolve` hook.

```
export async function dynamicInstantiate(url) {
  return {
    exports: ['customExportName'],
    execute: (exports) => {
      // get and set functions provided for pre-allocated export names
      exports.customExportName.set('value');
    }
  };
}
```

With the list of module exports provided upfront, the `execute` function will then be called at the exact point of module evaluation order for that module in the import tree.

Errors

Applications running in Node.js will generally experience four categories of errors:

- Standard JavaScript errors such as:
  - <EvalError> : thrown when a call to `eval()` fails.
  - <SyntaxError> : thrown in response to improper JavaScript language syntax.
  - <RangeError> : thrown when a value is not within an expected range
  - <ReferenceError> : thrown when using undefined variables
  - <TypeError> : thrown when passing arguments of the wrong type
  - <URIError> : thrown when a global URI handling function is misused.
- System errors triggered by underlying operating system constraints such as attempting to open a file that does not exist, attempting to send data over a closed socket, etc;
- And User-specified errors triggered by application code.
- Assertion Errors are a special class of error that can be triggered whenever Node.js detects an exceptional logic violation that should never occur. These are raised typically by the `assert` module.

All JavaScript and System errors raised by Node.js inherit from, or are instances of, the standard JavaScript <Error> class and are guaranteed to provide *at least* the properties available on that class.

Error Propagation and Interception

Node.js supports several mechanisms for propagating and handling errors that occur while an application is running. How these errors are reported and handled depends entirely on the type of Error and the style of the API that is called.

All JavaScript errors are handled as exceptions that *immediately* generate and throw an error using the standard JavaScript `throw` mechanism. These are handled using the try / catch construct provided by the JavaScript language.

```
// Throws with a ReferenceError because z is undefined
try {
  const m = 1;
  const n = m + z;
} catch (err) {
  // Handle the error here.
}
```

Any use of the JavaScript `throw` mechanism will raise an exception that *must* be handled using `try / catch` or the Node.js process will exit immediately.

With few exceptions, *Synchronous* APIs (any blocking method that does not accept a `callback` function, such as fs.readFileSync), will use `throw` to report errors.

Errors that occur within *Asynchronous APIs* may be reported in multiple ways:

- Most asynchronous methods that accept a `callback` function will accept an `Error` object passed as the first argument to that function. If that first argument is not `null` and is an instance of `Error`, then an error occurred that should be handled.

```
const fs = require('fs');
fs.readFile('a file that does not exist', (err, data) => {
  if (err) {
    console.error('There was an error reading the file!', err);
    return;
  }
  // Otherwise handle the data
});
```

- When an asynchronous method is called on an object that is an `EventEmitter`, errors can be routed to that object's `'error'` event.

```
const net = require('net');
const connection = net.connect('localhost');

// Adding an 'error' event handler to a stream:
connection.on('error', (err) => {
  // If the connection is reset by the server, or if it can't
  // connect at all, or on any sort of error encountered by
  // the connection, the error will be sent here.
  console.error(err);
});

connection.pipe(process.stdout);
```

- A handful of typically asynchronous methods in the Node.js API may still use the `throw` mechanism to raise exceptions that must be handled using `try / catch`. There is no comprehensive list of such methods; please refer to the documentation of each method to determine the appropriate error handling mechanism required.

The use of the `'error'` event mechanism is most common for stream-based and event emitter-based APIs, which themselves represent a series of asynchronous operations over time (as opposed to a single operation that may pass or fail).

For *all* `EventEmitter` objects, if an `'error'` event handler is not provided, the error will be thrown, causing the Node.js process to report an unhandled exception and crash unless either: The `domain` module is used appropriately or a handler has been registered for the `process.on('uncaughtException')` event.

```
const EventEmitter = require('events');
const ee = new EventEmitter();

setImmediate(() => {
  // This will crash the process because no 'error' event
  // handler has been added.
  ee.emit('error', new Error('This will crash'));
});
```

Errors generated in this way *cannot* be intercepted using `try / catch` as they are thrown *after* the calling code has already exited.

Developers must refer to the documentation for each method to determine exactly how errors raised by those methods are propagated.

Error-first callbacks

Most asynchronous methods exposed by the Node.js core API follow an idiomatic pattern referred to as an *error-first callback* (sometimes referred to as a *Node.js style callback*). With this pattern, a callback function is passed to the method as an argument. When the operation either completes or an error is raised, the callback function is called with the Error object (if any) passed as the first argument. If no error was raised, the first argument will be passed as `null`.

```
const fs = require('fs');

function errorFirstCallback(err, data) {
  if (err) {
    console.error('There was an error', err);
    return;
  }
  console.log(data);
}

fs.readFile('/some/file/that/does-not-exist', errorFirstCallback);
fs.readFile('/some/file/that/does-exist', errorFirstCallback);
```

The JavaScript `try / catch` mechanism **cannot** be used to intercept errors generated by asynchronous APIs. A common mistake for beginners is to try to use `throw` inside an error-first callback:

```
// THIS WILL NOT WORK:
const fs = require('fs');

try {
  fs.readFile('/some/file/that/does-not-exist', (err, data) => {
    // mistaken assumption: throwing here...
    if (err) {
      throw err;
    }
  });
} catch (err) {
  // This will not catch the throw!
  console.error(err);
}
```

This will not work because the callback function passed to `fs.readFile()` is called asynchronously. By the time the callback has been called, the surrounding code (including the `try { } catch (err) { }` block will have already exited. Throwing an error inside the callback **can crash the Node.js process** in most cases. If domains are enabled, or a handler has been registered with `process.on('uncaughtException')`, such errors can be intercepted.

Class: Error

A generic JavaScript `Error` object that does not denote any specific circumstance of why the error occurred. `Error` objects capture a "stack trace" detailing the point in the code at which the `Error` was instantiated, and may provide a text description of the error.

For crypto only, `Error` objects will include the OpenSSL error stack in a separate property called `opensslErrorStack` if it is available when the error is thrown.

All errors generated by Node.js, including all System and JavaScript errors, will either be instances of, or inherit from, the `Error` class.

new Error(message)

- `message` <string>

Creates a new `Error` object and sets the `error.message` property to the provided text message. If an object is passed as `message`, the text message is generated by calling `message.toString()`. The `error.stack` property will represent the point in the code at which `new Error()` was called. Stack traces are dependent on V8's stack trace API. Stack traces extend only to either (a) the beginning of *synchronous code execution*, or (b) the number of frames given by the property `Error.stackTraceLimit`, whichever is smaller.

Error.captureStackTrace(targetObject[, constructorOpt])

- `targetObject` <Object>
- `constructorOpt` <Function>

Creates a `.stack` property on `targetObject`, which when accessed returns a string representing the location in the code at which `Error.captureStackTrace()` was called.

```
const myObject = {};
Error.captureStackTrace(myObject);
myObject.stack;  // similar to `new Error().stack`
```

The first line of the trace will be prefixed with `${myObject.name}: ${myObject.message}`.

The optional `constructorOpt` argument accepts a function. If given, all frames above `constructorOpt`, including `constructorOpt`, will be omitted from the generated stack trace.

The `constructorOpt` argument is useful for hiding implementation details of error generation from an end user. For instance:

```
function MyError() {
  Error.captureStackTrace(this, MyError);
}

// Without passing MyError to captureStackTrace, the MyError
// frame would show up in the .stack property. By passing
// the constructor, we omit that frame, and retain all frames below it.
new MyError().stack;
```

Error.stackTraceLimit

- <number>

The `Error.stackTraceLimit` property specifies the number of stack frames collected by a stack trace (whether generated by `new Error().stack` or `Error.captureStackTrace(obj)`).

The default value is `10` but may be set to any valid JavaScript number. Changes will affect any stack trace captured*after* the value has been changed.

If set to a non-number value, or set to a negative number, stack traces will not capture any frames.

error.code

- <string>

The `error.code` property is a string label that identifies the kind of error. See Node.js Error Codes for details about specific codes.

error.message

- <string>

The `error.message` property is the string description of the error as set by calling `new Error(message)`. The `message` passed to the constructor will also appear in the first line of the stack trace of the `Error`, however changing this property after the `Error` object is created *may not* change the first line of the stack trace (for example, when `error.stack` is read before this property is changed).

```
const err = new Error('The message');
console.error(err.message);
// Prints: The message
```

error.stack

* <string>

The `error.stack` property is a string describing the point in the code at which the `Error` was instantiated.

```
Error: Things keep happening!
   at /home/gbusey/file.js:525:2
   at Frobnicator.refrobulate (/home/gbusey/business-logic.js:424:21)
   at Actor.<anonymous> (/home/gbusey/actors.js:400:8)
   at increaseSynergy (/home/gbusey/actors.js:701:6)
```

The first line is formatted as `<error class name>: <error message>`, and is followed by a series of stack frames (each line beginning with "at "). Each frame describes a call site within the code that lead to the error being generated. V8 attempts to display a name for each function (by variable name, function name, or object method name), but occasionally it will not be able to find a suitable name. If V8 cannot determine a name for the function, only location information will be displayed for that frame. Otherwise, the determined function name will be displayed with location information appended in parentheses.

Frames are only generated for JavaScript functions. If, for example, execution synchronously passes through a C++ addon function called `cheetahify` which itself calls a JavaScript function, the frame representing the `cheetahify` call will not be present in the stack traces:

```
const cheetahify = require('./native-binding.node');

function makeFaster() {
  // cheetahify *synchronously* calls speedy.
  cheetahify(function speedy() {
    throw new Error('oh no!');
  });
}

makeFaster();
// will throw:
//   /home/gbusey/file.js:6
//       throw new Error('oh no!');
//           ^
//   Error: oh no!
//       at speedy (/home/gbusey/file.js:6:11)
//       at makeFaster (/home/gbusey/file.js:5:3)
//       at Object.<anonymous> (/home/gbusey/file.js:10:1)
//       at Module._compile (module.js:456:26)
//       at Object.Module._extensions..js (module.js:474:10)
//       at Module.load (module.js:356:32)
//       at Function.Module._load (module.js:312:12)
//       at Function.Module.runMain (module.js:497:10)
//       at startup (node.js:119:16)
//       at node.js:906:3
```

The location information will be one of:

* `native`, if the frame represents a call internal to V8 (as in `[].forEach`).
* `plain-filename.js:line:column`, if the frame represents a call internal to Node.js.
* `/absolute/path/to/file.js:line:column`, if the frame represents a call in a user program, or its dependencies.

The string representing the stack trace is lazily generated when the `error.stack` property is **accessed**.

The number of frames captured by the stack trace is bounded by the smaller of `Error.stackTraceLimit` or the number of available frames on the current event loop tick.

System-level errors are generated as augmented `Error` instances, which are detailed here.

Class: AssertionError

A subclass of `Error` that indicates the failure of an assertion. Such errors commonly indicate inequality of actual and expected value.

```
assert.strictEqual(1, 2);
// AssertionError [ERR_ASSERTION]: 1 === 2
```

Class: RangeError

A subclass of `Error` that indicates that a provided argument was not within the set or range of acceptable values for a function; whether that is a numeric range, or outside the set of options for a given function parameter.

```
require('net').connect(-1);
// throws "RangeError: "port" option should be >= 0 and < 65536: -1"
```

Node.js will generate and throw `RangeError` instances *immediately* as a form of argument validation.

Class: ReferenceError

A subclass of `Error` that indicates that an attempt is being made to access a variable that is not defined. Such errors commonly indicate typos in code, or an otherwise broken program.

While client code may generate and propagate these errors, in practice, only V8 will do so.

```
doesNotExist;
// throws ReferenceError, doesNotExist is not a variable in this program.
```

Unless an application is dynamically generating and running code, `ReferenceError` instances should always be considered a bug in the code or its dependencies.

Class: SyntaxError

A subclass of `Error` that indicates that a program is not valid JavaScript. These errors may only be generated and propagated as a result of code evaluation. Code evaluation may happen as a result of `eval`, `Function`, `require`, or `vm`. These errors are almost always indicative of a broken program.

```
try {
  require('vm').runInThisContext('binary ! isNotOk');
} catch (err) {
  // err will be a SyntaxError
}
```

`SyntaxError` instances are unrecoverable in the context that created them – they may only be caught by other contexts.

Class: TypeError

A subclass of `Error` that indicates that a provided argument is not an allowable type. For example, passing a function to a parameter which expects a string would be considered a TypeError.

```
require('url').parse(() => { });
// throws TypeError, since it expected a string
```

Node.js will generate and throw `TypeError` instances *immediately* as a form of argument validation.

Exceptions vs. Errors

A JavaScript exception is a value that is thrown as a result of an invalid operation or as the target of a `throw` statement. While it is not required that these values are instances of `Error` or classes which inherit from `Error`, all exceptions thrown by Node.js or the JavaScript runtime *will* be instances of Error.

Some exceptions are *unrecoverable* at the JavaScript layer. Such exceptions will *always* cause the Node.js process to crash. Examples include `assert()` checks or `abort()` calls in the C++ layer.

System Errors

System errors are generated when exceptions occur within the program's runtime environment. Typically, these are operational errors that occur when an application violates an operating system constraint such as attempting to read a file that does not exist or when the user does not have sufficient permissions.

System errors are typically generated at the syscall level: an exhaustive list of error codes and their meanings is available by running `man 2 intro` or `man 3 errno` on most Unices; or online.

In Node.js, system errors are represented as augmented `Error` objects with added properties.

Class: System Error

error.code

- <string>

The `error.code` property is a string representing the error code, which is typically `E` followed by a sequence of capital letters.

error.errno

- <string> | <number>

The `error.errno` property is a number or a string. The number is a **negative** value which corresponds to the error code defined in libuv Error handling. See uv-errno.h header file (`deps/uv/include/uv-errno.h` in the Node.js source tree) for details. In case of a string, it is the same as `error.code`.

error.syscall

- <string>

The `error.syscall` property is a string describing the syscall that failed.

error.path

- <string>

When present (e.g. in `fs` or `child_process`), the `error.path` property is a string containing a relevant invalid pathname.

error.address

- <string>

When present (e.g. in `net` or `dgram`), the `error.address` property is a string describing the address to which the connection failed.

error.port

- <number>

When present (e.g. in `net` or `dgram`), the `error.port` property is a number representing the connection's port that is not available.

Common System Errors

This list is **not exhaustive**, but enumerates many of the common system errors encountered when writing a Node.js program. An exhaustive

list may be found here.

- `EACCES` (Permission denied): An attempt was made to access a file in a way forbidden by its file access permissions.
- `EADDRINUSE` (Address already in use): An attempt to bind a server (net, http, or https) to a local address failed due to another server on the local system already occupying that address.
- `ECONNREFUSED` (Connection refused): No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
- `ECONNRESET` (Connection reset by peer): A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or reboot. Commonly encountered via the http and net modules.
- `EEXIST` (File exists): An existing file was the target of an operation that required that the target not exist.
- `EISDIR` (Is a directory): An operation expected a file, but the given pathname was a directory.
- `EMFILE` (Too many open files in system): Maximum number of file descriptors allowable on the system has been reached, and requests for another descriptor cannot be fulfilled until at least one has been closed. This is encountered when opening many files at once in parallel, especially on systems (in particular, macOS) where there is a low file descriptor limit for processes. To remedy a low limit, run `ulimit -n 2048` in the same shell that will run the Node.js process.
- `ENOENT` (No such file or directory): Commonly raised by fs operations to indicate that a component of the specified pathname does not exist — no entity (file or directory) could be found by the given path.
- `ENOTDIR` (Not a directory): A component of the given pathname existed, but was not a directory as expected. Commonly raised by fs.readdir.
- `ENOTEMPTY` (Directory not empty): A directory with entries was the target of an operation that requires an empty directory — usually fs.unlink.
- `EPERM` (Operation not permitted): An attempt was made to perform an operation that requires elevated privileges.
- `EPIPE` (Broken pipe): A write on a pipe, socket, or FIFO for which there is no process to read the data. Commonly encountered at the net and http layers, indicative that the remote side of the stream being written to has been closed.
- `ETIMEDOUT` (Operation timed out): A connect or send request failed because the connected party did not properly respond after a period of time. Usually encountered by http or net — often a sign that a `socket.end()` was not properly called.

Node.js Error Codes

ERR_ARG_NOT_ITERABLE

An iterable argument (i.e. a value that works with `for...of` loops) was required, but not provided to a Node.js API.

ERR_ASSERTION

A special type of error that can be triggered whenever Node.js detects an exceptional logic violation that should never occur. These are raised typically by the `assert` module.

ERR_ASYNC_CALLBACK

An attempt was made to register something that is not a function as an `AsyncHooks` callback.

ERR_ASYNC_TYPE

The type of an asynchronous resource was invalid. Note that users are also able to define their own types if using the public embedder API.

ERR_BUFFER_OUT_OF_BOUNDS

An operation outside the bounds of a `Buffer` was attempted.

ERR_BUFFER_TOO_LARGE

An attempt has been made to create a `Buffer` larger than the maximum allowed size.

ERR_CHILD_CLOSED_BEFORE_REPLY

A child process was closed before the parent received a reply.

ERR_CONSOLE_WRITABLE_STREAM

`Console` was instantiated without `stdout` stream, or `Console` has a non-writable `stdout` or `stderr` stream.

ERR_CPU_USAGE

The native call from `process.cpuUsage` could not be processed.

ERR_CRYPTO_CUSTOM_ENGINE_NOT_SUPPORTED

A client certificate engine was requested that is not supported by the version of OpenSSL being used.

ERR_CRYPTO_ECDH_INVALID_FORMAT

An invalid value for the `format` argument was passed to the `crypto.ECDH()` class `getPublicKey()` method.

ERR_CRYPTO_ENGINE_UNKNOWN

An invalid crypto engine identifier was passed to `require('crypto').setEngine()` .

ERR_CRYPTO_FIPS_FORCED

The `--force-fips` command-line argument was used but there was an attempt to enable or disable FIPS mode in the `crypto` module.

ERR_CRYPTO_FIPS_UNAVAILABLE

An attempt was made to enable or disable FIPS mode, but FIPS mode was not available.

ERR_CRYPTO_HASH_DIGEST_NO_UTF16

The UTF-16 encoding was used with `hash.digest()`. While the `hash.digest()` method does allow an `encoding` argument to be passed in, causing the method to return a string rather than a `Buffer`, the UTF-16 encoding (e.g. `ucs` or `utf16le`) is not supported.

ERR_CRYPTO_HASH_FINALIZED

`hash.digest()` was called multiple times. The `hash.digest()` method must be called no more than one time per instance of a `Hash` object.

ERR_CRYPTO_HASH_UPDATE_FAILED

`hash.update()` failed for any reason. This should rarely, if ever, happen.

ERR_CRYPTO_INVALID_DIGEST

An invalid crypto digest algorithm was specified.

ERR_CRYPTO_SIGN_KEY_REQUIRED

A signing `key` was not provided to the `sign.sign()` method.

ERR_CRYPTO_TIMING_SAFE_EQUAL_LENGTH

`crypto.timingSafeEqual()` was called with `Buffer`, `TypedArray`, or `DataView` arguments of different lengths.

ERR_DNS_SET_SERVERS_FAILED

`c-ares` failed to set the DNS server.

ERR_DOMAIN_CALLBACK_NOT_AVAILABLE

The `domain` module was not usable since it could not establish the required error handling hooks, because `process.setUncaughtExceptionCaptureCallback()` had been called at an earlier point in time.

ERR_DOMAIN_CANNOT_SET_UNCAUGHT_EXCEPTION_CAPTURE

`process.setUncaughtExceptionCaptureCallback()` could not be called because the `domain` module has been loaded at an earlier point in time.

The stack trace is extended to include the point in time at which the `domain` module had been loaded.

ERR_ENCODING_INVALID_ENCODED_DATA

Data provided to `util.TextDecoder()` API was invalid according to the encoding provided.

ERR_ENCODING_NOT_SUPPORTED

Encoding provided to `util.TextDecoder()` API was not one of the WHATWG Supported Encodings .

ERR_FALSY_VALUE_REJECTION

A `Promise` that was callbackified via `util.callbackify()` was rejected with a falsy value.

ERR_HTTP_HEADERS_SENT

An attempt was made to add more headers after the headers had already been sent.

ERR_HTTP_INVALID_STATUS_CODE

Status code was outside the regular status code range (100-999).

ERR_HTTP_TRAILER_INVALID

The `Trailer` header was set even though the transfer encoding does not support that.

ERR_HTTP2_ALTSVC_INVALID_ORIGIN

HTTP/2 ALTSVC frames require a valid origin.

ERR_HTTP2_ALTSVC_LENGTH

HTTP/2 ALTSVC frames are limited to a maximum of 16,382 payload bytes.

ERR_HTTP2_CONNECT_AUTHORITY

For HTTP/2 requests using the `CONNECT` method, the `:authority` pseudo-header is required.

ERR_HTTP2_CONNECT_PATH

For HTTP/2 requests using the `CONNECT` method, the `:path` pseudo-header is forbidden.

ERR_HTTP2_CONNECT_SCHEME

For HTTP/2 requests using the `CONNECT` method, the `:scheme` pseudo-header is forbidden.

ERR_HTTP2_GOAWAY_SESSION

New HTTP/2 Streams may not be opened after the `Http2Session` has received a `GOAWAY` frame from the connected peer.

ERR_HTTP2_HEADER_SINGLE_VALUE

Multiple values were provided for an HTTP/2 header field that was required to have only a single value.

ERR_HTTP2_HEADERS_AFTER_RESPOND

An additional headers was specified after an HTTP/2 response was initiated.

ERR_HTTP2_HEADERS_SENT

An attempt was made to send multiple response headers.

ERR_HTTP2_INFO_STATUS_NOT_ALLOWED

Informational HTTP status codes (`1xx`) may not be set as the response status code on HTTP/2 responses.

ERR_HTTP2_INVALID_CONNECTION_HEADERS

HTTP/1 connection specific headers are forbidden to be used in HTTP/2 requests and responses.

ERR_HTTP2_INVALID_HEADER_VALUE

An invalid HTTP/2 header value was specified.

ERR_HTTP2_INVALID_INFO_STATUS

An invalid HTTP informational status code has been specified. Informational status codes must be an integer between `100` and `199` (inclusive).

ERR_HTTP2_INVALID_PACKED_SETTINGS_LENGTH

Input `Buffer` and `Uint8Array` instances passed to the `http2.getUnpackedSettings()` API must have a length that is a multiple of six.

ERR_HTTP2_INVALID_PSEUDOHEADER

Only valid HTTP/2 pseudoheaders (`:status` , `:path` , `:authority` , `:scheme` , and `:method`) may be used.

ERR_HTTP2_INVALID_SESSION

An action was performed on an `Http2Session` object that had already been destroyed.

ERR_HTTP2_INVALID_SETTING_VALUE

An invalid value has been specified for an HTTP/2 setting.

ERR_HTTP2_INVALID_STREAM

An operation was performed on a stream that had already been destroyed.

ERR_HTTP2_MAX_PENDING_SETTINGS_ACK

Whenever an HTTP/2 `SETTINGS` frame is sent to a connected peer, the peer is required to send an acknowledgment that it has received and applied the new `SETTINGS`. By default, a maximum number of unacknowledged `SETTINGS` frames may be sent at any given time. This error code is used when that limit has been reached.

ERR_HTTP2_NO_SOCKET_MANIPULATION

An attempt was made to directly manipulate (read, write, pause, resume, etc.) a socket attached to an `Http2Session`.

ERR_HTTP2_OUT_OF_STREAMS

The number of streams created on a single HTTP/2 session reached the maximum limit.

ERR_HTTP2_PAYLOAD_FORBIDDEN

A message payload was specified for an HTTP response code for which a payload is forbidden.

ERR_HTTP2_PING_CANCEL

An HTTP/2 ping was canceled.

ERR_HTTP2_PING_LENGTH

HTTP/2 ping payloads must be exactly 8 bytes in length.

ERR_HTTP2_PSEUDOHEADER_NOT_ALLOWED

An HTTP/2 pseudo-header has been used inappropriately. Pseudo-headers are header key names that begin with the `:` prefix.

ERR_HTTP2_PUSH_DISABLED

An attempt was made to create a push stream, which had been disabled by the client.

ERR_HTTP2_SEND_FILE

An attempt was made to use the `Http2Stream.prototype.responseWithFile()` API to send something other than a regular file.

ERR_HTTP2_SESSION_ERROR

The `Http2Session` closed with a non-zero error code.

ERR_HTTP2_SOCKET_BOUND

An attempt was made to connect a `Http2Session` object to a `net.Socket` or `tls.TLSSocket` that had already been bound to another `Http2Session` object.

ERR_HTTP2_STATUS_101

Use of the `101` Informational status code is forbidden in HTTP/2.

ERR_HTTP2_STATUS_INVALID

An invalid HTTP status code has been specified. Status codes must be an integer between `100` and `599` (inclusive).

ERR_HTTP2_STREAM_CANCEL

An `Http2Stream` was destroyed before any data was transmitted to the connected peer.

ERR_HTTP2_STREAM_ERROR

A non-zero error code was been specified in an `RST_STREAM` frame.

ERR_HTTP2_STREAM_SELF_DEPENDENCY

When setting the priority for an HTTP/2 stream, the stream may be marked as a dependency for a parent stream. This error code is used when an attempt is made to mark a stream and dependent of itself.

ERR_HTTP2_UNSUPPORTED_PROTOCOL

`http2.connect()` was passed a URL that uses any protocol other than `http:` or `https:`.

ERR_INDEX_OUT_OF_RANGE

A given index was out of the accepted range (e.g. negative offsets).

ERR_INSPECTOR_ALREADY_CONNECTED

While using the `inspector` module, an attempt was made to connect when the inspector was already connected.

ERR_INSPECTOR_CLOSED

While using the `inspector` module, an attempt was made to use the inspector after the session had already closed.

ERR_INSPECTOR_NOT_AVAILABLE

The `inspector` module is not available for use.

ERR_INSPECTOR_NOT_CONNECTED

While using the `inspector` module, an attempt was made to use the inspector before it was connected.

ERR_INVALID_ARG_TYPE

An argument of the wrong type was passed to a Node.js API.

ERR_INVALID_ARG_VALUE

An invalid or unsupported value was passed for a given argument.

ERR_INVALID_ARRAY_LENGTH

An Array was not of the expected length or in a valid range.

ERR_INVALID_ASYNC_ID

An invalid `asyncId` or `triggerAsyncId` was passed using `AsyncHooks`. An id less than -1 should never happen.

ERR_INVALID_BUFFER_SIZE

A swap was performed on a `Buffer` but its size was not compatible with the operation.

ERR_INVALID_CALLBACK

A callback function was required but was not been provided to a Node.js API.

ERR_INVALID_CHAR

Invalid characters were detected in headers.

ERR_INVALID_CURSOR_POS

A cursor on a given stream cannot be moved to a specified row without a specified column.

ERR_INVALID_DOMAIN_NAME

`hostname` can not be parsed from a provided URL.

ERR_INVALID_FD

A file descriptor ('fd') was not valid (e.g. it was a negative value).

ERR_INVALID_FD_TYPE

A file descriptor ('fd') type was not valid.

ERR_INVALID_FILE_URL_HOST

A Node.js API that consumes `file:` URLs (such as certain functions in the `fs` module) encountered a file URL with an incompatible host. This situation can only occur on Unix-like systems where only `localhost` or an empty host is supported.

ERR_INVALID_FILE_URL_PATH

A Node.js API that consumes `file:` URLs (such as certain functions in the `fs` module) encountered a file URL with an incompatible path. The exact semantics for determining whether a path can be used is platform-dependent.

ERR_INVALID_HANDLE_TYPE

An attempt was made to send an unsupported "handle" over an IPC communication channel to a child process. See `subprocess.send()` and `process.send()` for more information.

ERR_INVALID_HTTP_TOKEN

An invalid HTTP token was supplied.

ERR_INVALID_IP_ADDRESS

An IP address is not valid.

ERR_INVALID_OPT_VALUE

An invalid or unexpected value was passed in an options object.

ERR_INVALID_OPT_VALUE_ENCODING

An invalid or unknown file encoding was passed.

ERR_INVALID_PERFORMANCE_MARK

While using the Performance Timing API (`perf_hooks`), a performance mark is invalid.

ERR_INVALID_PROTOCOL

An invalid `options.protocol` was passed.

ERR_INVALID_REPL_EVAL_CONFIG

Both `breakEvalOnSigint` and `eval` options were set in the REPL config, which is not supported.

ERR_INVALID_SYNC_FORK_INPUT

A `Buffer`, `Uint8Array` or `string` was provided as stdio input to a synchronous fork. See the documentation for the `child_process` module for more information.

ERR_INVALID_THIS

A Node.js API function was called with an incompatible `this` value.

Example:

```
const { URLSearchParams } = require('url');
const urlSearchParams = new URLSearchParams('foo=bar&baz=new');

const buf = Buffer.alloc(1);
urlSearchParams.has.call(buf, 'foo');
// Throws a TypeError with code 'ERR_INVALID_THIS'
```

ERR_INVALID_TUPLE

An element in the iterable provided to the WHATWG URLSearchParams constructor did not represent a [name, value] tuple – that is, if an element is not iterable, or does not consist of exactly two elements.

ERR_INVALID_URI

An invalid URI was passed.

ERR_INVALID_URL

An invalid URL was passed to the WHATWG URL constructor to be parsed. The thrown error object typically has an additional property 'input' that contains the URL that failed to parse.

ERR_INVALID_URL_SCHEME

An attempt was made to use a URL of an incompatible scheme (protocol) for a specific purpose. It is only used in the WHATWG URL API support in the fs module (which only accepts URLs with 'file' scheme), but may be used in other Node.js APIs as well in the future.

ERR_IPC_CHANNEL_CLOSED

An attempt was made to use an IPC communication channel that was already closed.

ERR_IPC_DISCONNECTED

An attempt was made to disconnect an IPC communication channel that was already disconnected. See the documentation for the child_process module for more information.

ERR_IPC_ONE_PIPE

An attempt was made to create a child Node.js process using more than one IPC communication channel. See the documentation for the child_process module for more information.

ERR_IPC_SYNC_FORK

An attempt was made to open an IPC communication channel with a synchronously forked Node.js process. See the documentation for the child_process module for more information.

ERR_METHOD_NOT_IMPLEMENTED

A method is required but not implemented.

ERR_MISSING_ARGS

A required argument of a Node.js API was not passed. This is only used for strict compliance with the API specification (which in some cases may accept `func(undefined)` but not `func()`). In most native Node.js APIs, `func(undefined)` and `func()` are treated identically, and the `ERR_INVALID_ARG_TYPE` error code may be used instead.

ERR_MISSING_MODULE

Stability: 1 - Experimental

An ES6 module could not be resolved.

ERR_MODULE_RESOLUTION_LEGACY

Stability: 1 - Experimental

A failure occurred resolving imports in an ES6 module.

ERR_MULTIPLE_CALLBACK

A callback was called more than once.

A callback is almost always meant to only be called once as the query can either be fulfilled or rejected but not both at the same time. The latter would be possible by calling a callback more than once.

ERR_NAPI_CONS_FUNCTION

While using `N-API`, a constructor passed was not a function.

ERR_NAPI_INVALID_DATAVIEW_ARGS

While calling `napi_create_dataview()`, a given `offset` was outside the bounds of the dataview or `offset + length` was larger than a length of given `buffer`.

ERR_NAPI_INVALID_TYPEDARRAY_ALIGNMENT

While calling `napi_create_typedarray()`, the provided `offset` was not a multiple of the element size.

ERR_NAPI_INVALID_TYPEDARRAY_LENGTH

While calling `napi_create_typedarray()`, `(length * size_of_element) + byte_offset` was larger than the length of given `buffer`.

ERR_NO_CRYPTO

An attempt was made to use crypto features while Node.js was not compiled with OpenSSL crypto support.

ERR_NO_ICU

An attempt was made to use features that require ICU , but Node.js was not compiled with ICU support.

ERR_NO_LONGER_SUPPORTED

A Node.js API was called in an unsupported manner, such as `Buffer.write(string, encoding, offset[, length])` .

ERR_OUT_OF_RANGE

An input argument value was outside an acceptable range.

ERR_REQUIRE_ESM

Stability: 1 - Experimental

An attempt was made to `require()` an ES6 module .

ERR_SERVER_ALREADY_LISTEN

The `server.listen()` method was called while a `net.Server` was already listening. This applies to all instances of `net.Server` , including HTTP, HTTPS, and HTTP/2 Server instances.

ERR_SOCKET_ALREADY_BOUND

An attempt was made to bind a socket that has already been bound.

ERR_SOCKET_BAD_BUFFER_SIZE

An invalid (negative) size was passed for either the `recvBufferSize` or `sendBufferSize` options in `dgram.createSocket()` .

ERR_SOCKET_BAD_PORT

An API function expecting a port > 0 and < 65536 received an invalid value.

ERR_SOCKET_BAD_TYPE

An API function expecting a socket type (`udp4` or `udp6`) received an invalid value.

ERR_SOCKET_BUFFER_SIZE

While using dgram.createSocket(), the size of the receive or send `Buffer` could not be determined.

ERR_SOCKET_CANNOT_SEND

Data could be sent on a socket.

ERR_SOCKET_CLOSED

An attempt was made to operate on an already closed socket.

ERR_SOCKET_DGRAM_NOT_RUNNING

A call was made and the UDP subsystem was not running.

ERR_STDERR_CLOSE

An attempt was made to close the `process.stderr` stream. By design, Node.js does not allow `stdout` or `stderr` streams to be closed by user code.

ERR_STDOUT_CLOSE

An attempt was made to close the `process.stdout` stream. By design, Node.js does not allow `stdout` or `stderr` streams to be closed by user code.

ERR_STREAM_CANNOT_PIPE

An attempt was made to call stream.pipe() on a Writable stream.

ERR_STREAM_NULL_VALUES

An attempt was made to call stream.write() with a `null` chunk.

ERR_STREAM_PUSH_AFTER_EOF

An attempt was made to call stream.push() after a `null` (EOF) had been pushed to the stream.

ERR_STREAM_READ_NOT_IMPLEMENTED

An attempt was made to use a readable stream that did not implement readable._read().

ERR_STREAM_UNSHIFT_AFTER_END_EVENT

An attempt was made to call `stream.unshift()` after the `end` event was emitted.

ERR_STREAM_WRAP

Prevents an abort if a string decoder was set on the Socket or if the decoder is in `objectMode`.

Example

```
const Socket = require('net').Socket;
const instance = new Socket();

instance.setEncoding('utf8');
```

ERR_STREAM_WRITE_AFTER_END

An attempt was made to call `stream.write()` after `stream.end()` has been called.

ERR_TLS_CERT_ALTNAME_INVALID

While using TLS, the hostname/IP of the peer did not match any of the subjectAltNames in its certificate.

ERR_TLS_DH_PARAM_SIZE

While using TLS, the parameter offered for the Diffie-Hellman (`DH`) key-agreement protocol is too small. By default, the key length must be greater than or equal to 1024 bits to avoid vulnerabilities, even though it is strongly recommended to use 2048 bits or larger for stronger security.

ERR_TLS_HANDSHAKE_TIMEOUT

A TLS/SSL handshake timed out. In this case, the server must also abort the connection.

ERR_TLS_REQUIRED_SERVER_NAME

While using TLS, the `server.addContext()` method was called without providing a hostname in the first parameter.

ERR_TLS_SESSION_ATTACK

An excessive amount of TLS renegotiations is detected, which is a potential vector for denial-of-service attacks.

ERR_TRANSFORM_ALREADY_TRANSFORMING

A Transform stream finished while it was still transforming.

ERR_TRANSFORM_WITH_LENGTH_0

A Transform stream finished with data still in the write buffer.

ERR_UNCAUGHT_EXCEPTION_CAPTURE_ALREADY_SET

`process.setUncaughtExceptionCaptureCallback()` was called twice, without first resetting the callback to `null`.

This error is designed to prevent accidentally overwriting a callback registered from another module.

ERR_UNESCAPED_CHARACTERS

A string that contained unescaped characters was received.

ERR_UNHANDLED_ERROR

An unhandled error occurred (for instance, when an `'error'` event is emitted by an `EventEmitter` but an `'error'` handler is not registered).

ERR_UNKNOWN_ENCODING

An invalid or unknown encoding option was passed to an API.

ERR_UNKNOWN_FILE_EXTENSION

Stability: 1 - Experimental

An attempt was made to load a module with an unknown or unsupported file extension.

ERR_UNKNOWN_MODULE_FORMAT

Stability: 1 - Experimental

An attempt was made to load a module with an unknown or unsupported format.

ERR_UNKNOWN_SIGNAL

An invalid or unknown process signal was passed to an API expecting a valid signal (such as `subprocess.kill()`).

ERR_UNKNOWN_STDIN_TYPE

An attempt was made to launch a Node.js process with an unknown `stdin` file type. This error is usually an indication of a bug within Node.js itself, although it is possible for user code to trigger it.

ERR_UNKNOWN_STREAM_TYPE

An attempt was made to launch a Node.js process with an unknown `stdout` or `stderr` file type. This error is usually an indication of a bug within Node.js itself, although it is possible for user code to trigger it.

ERR_V8BREAKITERATOR

The V8 BreakIterator API was used but the full ICU data set is not installed.

ERR_VALID_PERFORMANCE_ENTRY_TYPE

While using the Performance Timing API (`perf_hooks`), no valid performance entry types were found.

ERR_VALUE_OUT_OF_RANGE

A given value is out of the accepted range.

ERR_VM_MODULE_ALREADY_LINKED

The module attempted to be linked is not eligible for linking, because of one of the following reasons:

- It has already been linked (`linkingStatus` is `'linked'`)
- It is being linked (`linkingStatus` is `'linking'`)
- Linking has failed for this module (`linkingStatus` is `'errored'`)

ERR_VM_MODULE_DIFFERENT_CONTEXT

The module being returned from the linker function is from a different context than the parent module. Linked modules must share the same context.

ERR_VM_MODULE_LINKING_ERRORED

The linker function returned a module for which linking has failed.

ERR_VM_MODULE_NOT_LINKED

The module must be successfully linked before instantiation.

ERR_VM_MODULE_NOT_MODULE

The fulfilled value of a linking promise is not a `vm.Module` object.

ERR_VM_MODULE_STATUS

The current module's status does not allow for this operation. The specific meaning of the error depends on the specific function.

ERR_ZLIB_BINDING_CLOSED

An attempt was made to use a `zlib` object after it has already been closed.

ERR_ZLIB_INITIALIZATION_FAILED

Creation of a zlib object failed due to incorrect configuration.

Events

Stability: 2 - Stable

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be called.

For instance: a net.Server object emits an event each time a peer connects to it; a fs.ReadStream emits an event when the file is opened; a stream emits an event whenever data is available to be read.

All objects that emit events are instances of the `EventEmitter` class. These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.

When the `EventEmitter` object emits an event, all of the functions attached to that specific event are called*synchronously*. Any values returned by the called listeners are *ignored* and will be discarded.

The following example shows a simple `EventEmitter` instance with a single listener. The `eventEmitter.on()` method is used to register listeners, while the `eventEmitter.emit()` method is used to trigger the event.

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

Passing arguments and `this` to listeners

The `eventEmitter.emit()` method allows an arbitrary set of arguments to be passed to the listener functions. It is important to keep in mind that when an ordinary listener function is called by the `EventEmitter`, the standard `this` keyword is intentionally set to reference the `EventEmitter` to which the listener is attached.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this);
  // Prints:
  //   a b MyEmitter {
  //     domain: null,
  //     _events: { event: [Function] },
  //     _eventsCount: 1,
  //     _maxListeners: undefined }
});
myEmitter.emit('event', 'a', 'b');
```

It is possible to use ES6 Arrow Functions as listeners, however, when doing so, the `this` keyword will no longer reference the `EventEmitter` instance:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  console.log(a, b, this);
  // Prints: a b {}
});
myEmitter.emit('event', 'a', 'b');
```

Asynchronous vs. Synchronous

The `EventEmitter` calls all listeners synchronously in the order in which they were registered. This is important to ensure the proper sequencing of events and to avoid race conditions or logic errors. When appropriate, listener functions can switch to an asynchronous mode of operation using the `setImmediate()` or `process.nextTick()` methods:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  setImmediate(() => {
    console.log('this happens asynchronously');
  });
});
myEmitter.emit('event', 'a', 'b');
```

Handling events only once

When a listener is registered using the `eventEmitter.on()` method, that listener will be invoked *every time* the named event is emitted.

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.on('event', () => {
  console.log(++m);
});
myEmitter.emit('event');
// Prints: 1
myEmitter.emit('event');
// Prints: 2
```

Using the `eventEmitter.once()` method, it is possible to register a listener that is called at most once for a particular event. Once the event is emitted, the listener is unregistered and *then* called.

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.once('event', () => {
  console.log(++m);
});
myEmitter.emit('event');
// Prints: 1
myEmitter.emit('event');
// Ignored
```

Error events

When an error occurs within an `EventEmitter` instance, the typical action is for an `'error'` event to be emitted. These are treated as special cases within Node.js.

If an `EventEmitter` does *not* have at least one listener registered for the `'error'` event, and an `'error'` event is emitted, the error is thrown, a stack trace is printed, and the Node.js process exits.

```
const myEmitter = new MyEmitter();
myEmitter.emit('error', new Error('whoops!'));
// Throws and crashes Node.js
```

To guard against crashing the Node.js process the domain module can be used. (Note, however, that the `domain` module has been deprecated.)

As a best practice, listeners should always be added for the `'error'` events.

```
const myEmitter = new MyEmitter();
myEmitter.on('error', (err) => {
  console.error('whoops! there was an error');
});
myEmitter.emit('error', new Error('whoops!'));
// Prints: whoops! there was an error
```

Class: EventEmitter

Added in: v0.1.26

The `EventEmitter` class is defined and exposed by the `events` module:

```
const EventEmitter = require('events');
```

All EventEmitters emit the event `'newListener'` when new listeners are added and `'removeListener'` when existing listeners are removed.

Event: 'newListener'

Added in: v0.1.26

- `eventName` <string> | <symbol> The name of the event being listened for
- `listener` <Function> The event handler function

The `EventEmitter` instance will emit its own `'newListener'` event *before* a listener is added to its internal array of listeners.

Listeners registered for the `'newListener'` event will be passed the event name and a reference to the listener being added.

The fact that the event is triggered before adding the listener has a subtle but important side effect: any*additional* listeners registered to the same `name` *within* the `'newListener'` callback will be inserted *before* the listener that is in the process of being added.

```
const myEmitter = new MyEmitter();
// Only do this once so we don't loop forever
myEmitter.once('newListener', (event, listener) => {
  if (event === 'event') {
    // Insert a new listener in front
    myEmitter.on('event', () => {
      console.log('B');
    });
  }
});
myEmitter.on('event', () => {
  console.log('A');
});
myEmitter.emit('event');
// Prints:
//   B
//   A
```

Event: 'removeListener'

▸ History

- `eventName` <string> | <symbol> The event name
- `listener` <Function> The event handler function

The `'removeListener'` event is emitted *after* the `listener` is removed.

EventEmitter.listenerCount(emitter, eventName)

Added in: v0.9.12   Deprecated since: v4.0.0

Stability: 0 - Deprecated: Use emitter.listenerCount() instead.

A class method that returns the number of listeners for the given `eventName` registered on the given `emitter`.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {});
myEmitter.on('event', () => {});
console.log(EventEmitter.listenerCount(myEmitter, 'event'));
// Prints: 2
```

EventEmitter.defaultMaxListeners

Added in: v0.11.2

By default, a maximum of `10` listeners can be registered for any single event. This limit can be changed for individual `EventEmitter` instances using the emitter.setMaxListeners(n) method. To change the default for *all* `EventEmitter` instances, the `EventEmitter.defaultMaxListeners` property can be used. If this value is not a positive number, a `TypeError` will be thrown.

Take caution when setting the `EventEmitter.defaultMaxListeners` because the change affects *all* `EventEmitter` instances, including those created before the change is made. However, calling emitter.setMaxListeners(n) still has precedence over `EventEmitter.defaultMaxListeners`.

Note that this is not a hard limit. The `EventEmitter` instance will allow more listeners to be added but will output a trace warning to stderr indicating that a "possible EventEmitter memory leak" has been detected. For any single `EventEmitter`, the `emitter.getMaxListeners()` and `emitter.setMaxListeners()` methods can be used to temporarily avoid this warning:

```
emitter.setMaxListeners(emitter.getMaxListeners() + 1);
emitter.once('event', () => {
  // do stuff
  emitter.setMaxListeners(Math.max(emitter.getMaxListeners() - 1, 0));
});
```

The `--trace-warnings` command line flag can be used to display the stack trace for such warnings.

The emitted warning can be inspected with `process.on('warning')` and will have the additional `emitter`, `type` and `count` properties, referring to the event emitter instance, the event's name and the number of attached listeners, respectively. Its `name` property is set to `'MaxListenersExceededWarning'`.

emitter.addListener(eventName, listener)

Added in: v0.1.26

- `eventName` <string> | <symbol>
- `listener` <Function>

Alias for `emitter.on(eventName, listener)`.

emitter.emit(eventName[, ...args])

Added in: v0.1.26

- `eventName` <string> | <symbol>
- `...args` <any>

Synchronously calls each of the listeners registered for the event named `eventName`, in the order they were registered, passing the supplied arguments to each.

Returns `true` if the event had listeners, `false` otherwise.

emitter.eventNames()

Added in: v6.0.0

Returns an array listing the events for which the emitter has registered listeners. The values in the array will be strings or Symbols.

```
const EventEmitter = require('events');
const myEE = new EventEmitter();
myEE.on('foo', () => {});
myEE.on('bar', () => {});

const sym = Symbol('symbol');
myEE.on(sym, () => {});

console.log(myEE.eventNames());
// Prints: [ 'foo', 'bar', Symbol(symbol) ]
```

emitter.getMaxListeners()

Added in: v1.0.0

Returns the current max listener value for the `EventEmitter` which is either set by `emitter.setMaxListeners(n)` or defaults to `EventEmitter.defaultMaxListeners`.

emitter.listenerCount(eventName)

Added in: v3.2.0

- `eventName` <string> | <symbol> The name of the event being listened for

Returns the number of listeners listening to the event named `eventName`.

emitter.listeners(eventName)

▸ History

- `eventName` <string> | <symbol>

Returns a copy of the array of listeners for the event named `eventName`.

```
server.on('connection', (stream) => {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection')));
// Prints: [ [Function] ]
```

emitter.on(eventName, listener)

Added in: v0.1.101

- `eventName` <string> | <symbol> The name of the event.
- `listener` <Function> The callback function

Adds the `listener` function to the end of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

```
server.on('connection', (stream) => {
  console.log('someone connected!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

By default, event listeners are invoked in the order they are added. The `emitter.prependListener()` method can be used as an alternative to add the event listener to the beginning of the listeners array.

```
const myEE = new EventEmitter();
myEE.on('foo', () => console.log('a'));
myEE.prependListener('foo', () => console.log('b'));
myEE.emit('foo');
// Prints:
//   b
//   a
```

emitter.once(eventName, listener)

Added in: v0.3.0

- `eventName` <string> | <symbol> The name of the event.
- `listener` <Function> The callback function

Adds a **one-time** `listener` function for the event named `eventName`. The next time `eventName` is triggered, this listener is removed and then invoked.

```
server.once('connection', (stream) => {
  console.log('Ah, we have our first user!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

By default, event listeners are invoked in the order they are added. The `emitter.prependOnceListener()` method can be used as an alternative to add the event listener to the beginning of the listeners array.

```
const myEE = new EventEmitter();
myEE.once('foo', () => console.log('a'));
myEE.prependOnceListener('foo', () => console.log('b'));
myEE.emit('foo');
// Prints:
//   b
//   a
```

emitter.prependListener(eventName, listener)

Added in: v6.0.0

- `eventName` <string> | <symbol> The name of the event.
- `listener` <Function> The callback function

Adds the `listener` function to the *beginning* of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

```
server.prependListener('connection', (stream) => {
  console.log('someone connected!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

emitter.prependOnceListener(eventName, listener)

Added in: v6.0.0

- `eventName` <string> | <symbol> The name of the event.
- `listener` <Function> The callback function

Adds a **one-time** `listener` function for the event named `eventName` to the *beginning* of the listeners array. The next time `eventName` is triggered, this listener is removed, and then invoked.

```
server.prependOnceListener('connection', (stream) => {
  console.log('Ah, we have our first user!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

emitter.removeAllListeners([eventName])

Added in: v0.1.26

- `eventName` <string> | <symbol>

Removes all listeners, or those of the specified `eventName`.

Note that it is bad practice to remove listeners added elsewhere in the code, particularly when the `EventEmitter` instance was created by some other component or module (e.g. sockets or file streams).

Returns a reference to the `EventEmitter`, so that calls can be chained.

emitter.removeListener(eventName, listener)

Added in: v0.1.26

- `eventName` <string> | <symbol>
- `listener` <Function>

Removes the specified `listener` from the listener array for the event named `eventName`.

```
const callback = (stream) => {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

`removeListener` will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified `eventName`, then `removeListener` must be called multiple times to remove each instance.

Note that once an event has been emitted, all listeners attached to it at the time of emitting will be called in order. This implies that any `removeListener()` or `removeAllListeners()` calls *after* emitting and *before* the last listener finishes execution will not remove them from `emit()` in progress. Subsequent events will behave as expected.

```
const myEmitter = new MyEmitter();

const callbackA = () => {
  console.log('A');
  myEmitter.removeListener('event', callbackB);
};

const callbackB = () => {
  console.log('B');
};

myEmitter.on('event', callbackA);

myEmitter.on('event', callbackB);

// callbackA removes listener callbackB but it will still be called.
// Internal listener array at time of emit [callbackA, callbackB]
myEmitter.emit('event');
// Prints:
//   A
//   B

// callbackB is now removed.
// Internal listener array [callbackA]
myEmitter.emit('event');
// Prints:
//   A
```

Because listeners are managed using an internal array, calling this will change the position indices of any listener registered *after* the listener being removed. This will not impact the order in which listeners are called, but it means that any copies of the listener array as returned by

the `emitter.listeners()` method will need to be recreated.

Returns a reference to the `EventEmitter`, so that calls can be chained.

emitter.setMaxListeners(n)

Added in: v0.3.5

- `n` <integer>

By default EventEmitters will print a warning if more than `10` listeners are added for a particular event. This is a useful default that helps finding memory leaks. Obviously, not all events should be limited to just 10 listeners. The `emitter.setMaxListeners()` method allows the limit to be modified for this specific `EventEmitter` instance. The value can be set to `Infinity` (or `0`) to indicate an unlimited number of listeners.

Returns a reference to the `EventEmitter`, so that calls can be chained.

emitter.rawListeners(eventName)

Added in: v9.4.0

- `eventName` <string> | <symbol>

Returns a copy of the array of listeners for the event named `eventName`, including any wrappers (such as those created by `.once`).

```
const emitter = new EventEmitter();
emitter.once('log', () => console.log('log once'));

// Returns a new Array with a function `onceWrapper` which has a property
// `listener` which contains the original listener bound above
const listeners = emitter.rawListeners('log');
const logFnWrapper = listeners[0];

// logs "log once" to the console and does not unbind the `once` event
logFnWrapper.listener();

// logs "log once" to the console and removes the listener
logFnWrapper();

emitter.on('log', () => console.log('log persistently'));
// will return a new Array with a single function bound by `on` above
const newListeners = emitter.rawListeners('log');

// logs "log persistently" twice
newListeners[0]();
emitter.emit('log');
```

File System

Stability: 2 - Stable

The `fs` module provides an API for interacting with the file system in a manner closely modeled around standard POSIX functions.

To use this module:

```
const fs = require('fs');
```

All file system operations have synchronous and asynchronous forms.

The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on

the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

```
const fs = require('fs');

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

Exceptions that occur using synchronous operations are thrown immediately and may be handled using `try` / `catch`, or may be allowed to bubble up.

```
const fs = require('fs');

try {
  fs.unlinkSync('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (err) {
  // handle the error
}
```

Note that there is no guaranteed ordering when using asynchronous methods. So the following is prone to error because the `fs.stat()` operation may complete before the `fs.rename()` operation.

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', (err, stats) => {
  if (err) throw err;
  console.log(`stats: ${JSON.stringify(stats)}`);
});
```

To correctly order the operations, move the `fs.stat()` call into the callback of the `fs.rename()` operation:

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  fs.stat('/tmp/world', (err, stats) => {
    if (err) throw err;
    console.log(`stats: ${JSON.stringify(stats)}`);
  });
});
```

In busy processes, the programmer is *strongly encouraged* to use the asynchronous versions of these calls. The synchronous versions will block the entire process until they complete — halting all connections.

While it is not recommended, most fs functions allow the callback argument to be omitted, in which case a default callback is used that rethrows errors. To get a trace to the original call site, set the `NODE_DEBUG` environment variable:

Omitting the callback function on asynchronous fs functions is deprecated and may result in an error being thrown in the future.

```
$ cat script.js
function bad() {
  require('fs').readFile('/');
}
bad();

$ env NODE_DEBUG=fs node script.js
fs.js:88
        throw backtrace;
        ^
Error: EISDIR: illegal operation on a directory, read
    <stack trace.>
```

File paths

Most `fs` operations accept filepaths that may be specified in the form of a string, a `Buffer`, or a `URL` object using the `file:` protocol.

String form paths are interpreted as UTF-8 character sequences identifying the absolute or relative filename. Relative paths will be resolved relative to the current working directory as specified by `process.cwd()`.

Example using an absolute path on POSIX:

```
const fs = require('fs');

fs.open('/open/some/file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});
```

Example using a relative path on POSIX (relative to `process.cwd()`):

```
fs.open('file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});
```

Paths specified using a `Buffer` are useful primarily on certain POSIX operating systems that treat file paths as opaque byte sequences. On such systems, it is possible for a single file path to contain sub-sequences that use multiple character encodings. As with string paths, `Buffer` paths may be relative or absolute:

Example using an absolute path on POSIX:

```
fs.open(Buffer.from('/open/some/file.txt'), 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});
```

Note: On Windows Node.js follows the concept of per-drive working directory. This behavior can be observed when using a drive path without a backslash. For example `fs.readdirSync('c:\\')` can potentially return a different result than `fs.readdirSync('c:')`. For more information,

see this MSDN page.

URL object support

Added in: v7.6.0

For most `fs` module functions, the `path` or `filename` argument may be passed as a WHATWG `URL` object. Only `URL` objects using the `file:` protocol are supported.

```
const fs = require('fs');
const { URL } = require('url');
const fileUrl = new URL('file:///tmp/hello');

fs.readFileSync(fileUrl);
```

`file:` URLs are always absolute paths.

Using WHATWG `URL` objects might introduce platform-specific behaviors.

On Windows, `file:` URLs with a hostname convert to UNC paths, while `file:` URLs with drive letters convert to local absolute paths. `file:` URLs without a hostname nor a drive letter will result in a throw :

```
// On Windows :

// - WHATWG file URLs with hostname convert to UNC path
// file://hostname/p/a/t/h/file => \\hostname\p\a\t\h\file
fs.readFileSync(new URL('file://hostname/p/a/t/h/file'));

// - WHATWG file URLs with drive letters convert to absolute path
// file:///C:/tmp/hello => C:\tmp\hello
fs.readFileSync(new URL('file:///C:/tmp/hello'));

// - WHATWG file URLs without hostname must have a drive letters
fs.readFileSync(new URL('file:///notdriveletter/p/a/t/h/file'));
fs.readFileSync(new URL('file:///c/p/a/t/h/file'));
// TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must be absolute
```

`file:` URLs with drive letters must use `:` as a separator just after the drive letter. Using another separator will result in a throw.

On all other platforms, `file:` URLs with a hostname are unsupported and will result in a throw:

```
// On other platforms:

// - WHATWG file URLs with hostname are unsupported
// file://hostname/p/a/t/h/file => throw!
fs.readFileSync(new URL('file://hostname/p/a/t/h/file'));
// TypeError [ERR_INVALID_FILE_URL_PATH]: must be absolute

// - WHATWG file URLs convert to absolute path
// file:///tmp/hello => /tmp/hello
fs.readFileSync(new URL('file:///tmp/hello'));
```

A `file:` URL having encoded slash characters will result in a throw on all platforms:

```
// On Windows
fs.readFileSync(new URL('file:///C:/p/a/t/h/%2F'));
fs.readFileSync(new URL('file:///C:/p/a/t/h/%2f'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
\ or / characters */

// On POSIX
fs.readFileSync(new URL('file:///p/a/t/h/%2F'));
fs.readFileSync(new URL('file:///p/a/t/h/%2f'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
/ characters */
```

On Windows, `file:` URLs having encoded backslash will result in a throw:

```
// On Windows
fs.readFileSync(new URL('file:///C:/path/%5C'));
fs.readFileSync(new URL('file:///C:/path/%5c'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
\ or / characters */
```

File Descriptors

On POSIX systems, for every process, the kernel maintains a table of currently open files and resources. Each open file is assigned a simple numeric identifier called a *file descriptor*. At the system-level, all file system operations use these file descriptors to identify and track each specific file. Windows systems use a different but conceptually similar mechanism for tracking resources. To simplify things for users, Node.js abstracts away the specific differences between operating systems and assigns all open files a numeric file descriptor.

The `fs.open()` method is used to allocate a new file descriptor. Once allocated, the file descriptor may be used to read data from, write data to, or request information about the file.

```
fs.open('/open/some/file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.fstat(fd, (err, stat) => {
    if (err) throw err;
    // use stat

    // always close the file descriptor!
    fs.close(fd, (err) => {
      if (err) throw err;
    });
  });
});
```

Most operating systems limit the number of file descriptors that may be open at any given time so it is critical to close the descriptor when operations are completed. Failure to do so will result in a memory leak that will eventually cause an application to crash.

Threadpool Usage

Note that all file system APIs except `fs.FSWatcher()` and those that are explicitly synchronous use libuv's threadpool, which can have surprising and negative performance implications for some applications, see the `UV_THREADPOOL_SIZE` documentation for more information.

Class: fs.FSWatcher

Added in: v0.5.8

A successful call to fs.watch() method will return a new fs.FSWatcher object.

All fs.FSWatcher objects are EventEmitter 's that will emit a 'change' event whenever a specific watched file is modified.

Event: 'change'

Added in: v0.5.8

- eventType <string> The type of change event that has occurred
- filename <string> | <Buffer> The filename that changed (if relevant/available)

Emitted when something changes in a watched directory or file. See more details in fs.watch() .

The filename argument may not be provided depending on operating system support. If filename is provided, it will be provided as a Buffer if fs.watch() is called with its encoding option set to 'buffer', otherwise filename will be a UTF-8 string.

```
// Example when handled through fs.watch listener
fs.watch('./tmp', { encoding: 'buffer' }, (eventType, filename) => {
  if (filename) {
    console.log(filename);
    // Prints: <Buffer ...>
  }
});
```

Event: 'error'

Added in: v0.5.8

- error <Error>

Emitted when an error occurs while watching the file.

watcher.close()

Added in: v0.5.8

Stop watching for changes on the given fs.FSWatcher . Once stopped, the fs.FSWatcher object is no longer usable.

Class: fs.ReadStream

Added in: v0.1.93

A successful call to fs.createReadStream() will return a new fs.ReadStream object.

All fs.ReadStream objects are Readable Streams .

Event: 'close'

Added in: v0.1.93

Emitted when the fs.ReadStream 's underlying file descriptor has been closed.

Event: 'open'

Added in: v0.1.93

- fd <integer> Integer file descriptor used by the ReadStream.

Emitted when the fs.ReadStream 's file descriptor has been opened.

readStream.bytesRead

Added in: v6.4.0

- Value: <number>

The number of bytes that have been read so far.

readStream.path

Added in: v0.1.93

- Value: <string> | <Buffer>

The path to the file the stream is reading from as specified in the first argument to `fs.createReadStream()`. If `path` is passed as a string, then `readStream.path` will be a string. If `path` is passed as a `Buffer`, then `readStream.path` will be a `Buffer`.

Class: fs.Stats

▸ History

A `fs.Stats` object provides information about a file.

Objects returned from fs.stat(), fs.lstat() and fs.fstat() and their synchronous counterparts are of this type.

```
Stats {
  dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atimeMs: 1318289051000.1,
  mtimeMs: 1318289051000.1,
  ctimeMs: 1318289051000.1,
  birthtimeMs: 1318289051000.1,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT,
  birthtime: Mon, 10 Oct 2011 23:24:11 GMT }
```

stats.isBlockDevice()

Added in: v0.1.10

- Returns: <boolean>

Returns `true` if the `fs.Stats` object describes a block device.

stats.isCharacterDevice()

Added in: v0.1.10

- Returns: <boolean>

Returns `true` if the `fs.Stats` object describes a character device.

stats.isDirectory()

Added in: v0.1.10

- Returns: <boolean>

Returns `true` if the `fs.Stats` object describes a file system directory.

stats.isFIFO()

Added in: v0.1.10

- Returns: <boolean>

Returns `true` if the `fs.Stats` object describes a first-in-first-out (FIFO) pipe.

stats.isFile()

Added in: v0.1.10

- Returns: <boolean>

Returns `true` if the `fs.Stats` object describes a regular file.

stats.isSocket()

Added in: v0.1.10

- Returns: <boolean>

Returns `true` if the `fs.Stats` object describes a socket.

stats.isSymbolicLink()

Added in: v0.1.10

- Returns: <boolean>

Returns `true` if the `fs.Stats` object describes a symbolic link.

This method is only valid when using fs.lstat()

stats.dev

- Value: <number>

The numeric identifier of the device containing the file.

stats.ino

- Value: <number>

The file system specific "Inode" number for the file.

stats.mode

- Value: <number>

A bit-field describing the file type and mode.

stats.nlink

- Value: <number>

The number of hard-links that exist for the file.

stats.uid

- Value: <number>

The numeric user identifier of the user that owns the file (POSIX).

stats.gid

- Value: <number>

The numeric group identifier of the group that owns the file (POSIX).

stats.rdev

- Value: <number>

A numeric device identifier if the file is considered "special".

stats.size

- Value: <number>

The size of the file in bytes.

stats.blksize

- Value: <number>

The file system block size for i/o operations.

stats.blocks

- Value: <number>

The number of blocks allocated for this file.

stats.atimeMs

Added in: v8.1.0

- Value: <number>

The timestamp indicating the last time this file was accessed expressed in milliseconds since the POSIX Epoch.

stats.mtimeMs

Added in: v8.1.0

- Value: <number>

The timestamp indicating the last time this file was modified expressed in milliseconds since the POSIX Epoch.

stats.ctimeMs

Added in: v8.1.0

- Value: <number>

The timestamp indicating the last time the file status was changed expressed in milliseconds since the POSIX Epoch.

stats.birthtimeMs

Added in: v8.1.0

- Value: <number>

The timestamp indicating the creation time of this file expressed in milliseconds since the POSIX Epoch.

stats.atime

Added in: v0.11.13

- Value: <Date>

The timestamp indicating the last time this file was accessed.

stats.mtime

Added in: v0.11.13

- Value: <Date>

The timestamp indicating the last time this file was modified.

stats.ctime

Added in: v0.11.13

- Value: <Date>

The timestamp indicating the last time the file status was changed.

stats.birthtime

Added in: v0.11.13

- Value: <Date>

The timestamp indicating the creation time of this file.

Stat Time Values

The `atimeMs`, `mtimeMs`, `ctimeMs`, `birthtimeMs` properties are numbers that hold the corresponding times in milliseconds. Their precision is platform specific. `atime`, `mtime`, `ctime`, and `birthtime` are Date object alternate representations of the various times. The `Date` and number values are not connected. Assigning a new number value, or mutating the `Date` value, will not be reflected in the corresponding alternate representation.

The times in the stat object have the following semantics:

- `atime` "Access Time" - Time when file data last accessed. Changed by the mknod(2), utimes(2), and read(2) system calls.
- `mtime` "Modified Time" - Time when file data last modified. Changed by the mknod(2), utimes(2), and write(2) system calls.
- `ctime` "Change Time" - Time when file status was last changed (inode data modification). Changed by the chmod(2), chown(2), link(2), mknod(2), rename(2), unlink(2), utimes(2), read(2), and write(2) system calls.
- `birthtime` "Birth Time" - Time of file creation. Set once when the file is created. On filesystems where birthtime is not available, this field may instead hold either the `ctime` or `1970-01-01T00:00Z` (ie, unix epoch timestamp `0`). Note that this value may be greater than `atime` or `mtime` in this case. On Darwin and other FreeBSD variants, also set if the `atime` is explicitly set to an earlier value than the current `birthtime` using the utimes(2) system call.

Prior to Node.js v0.12, the `ctime` held the `birthtime` on Windows systems. Note that as of v0.12, `ctime` is not "creation time", and on Unix systems, it never was.

Class: fs.WriteStream

Added in: v0.1.93

`WriteStream` is a Writable Stream.

Event: 'close'

Added in: v0.1.93

Emitted when the `WriteStream`'s underlying file descriptor has been closed.

Event: 'open'

Added in: v0.1.93

- `fd` <integer> Integer file descriptor used by the WriteStream.

Emitted when the WriteStream's file is opened.

writeStream.bytesWritten

Added in: v0.4.7

The number of bytes written so far. Does not include data that is still queued for writing.

writeStream.path

Added in: v0.1.93

The path to the file the stream is writing to as specified in the first argument to `fs.createWriteStream()`. If `path` is passed as a string, then `writeStream.path` will be a string. If `path` is passed as a `Buffer`, then `writeStream.path` will be a `Buffer`.

fs.access(path[, mode], callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `mode` <integer> **Default:** `fs.constants.F_OK`
- `callback` <Function>
  - `err` <Error>

Tests a user's permissions for the file or directory specified by `path`. The `mode` argument is an optional integer that specifies the accessibility checks to be performed. The following constants define the possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.W_OK | fs.constants.R_OK`).

- `fs.constants.F_OK` - `path` is visible to the calling process. This is useful for determining if a file exists, but says nothing about `rwx` permissions. Default if no `mode` is specified.
- `fs.constants.R_OK` - `path` can be read by the calling process.
- `fs.constants.W_OK` - `path` can be written by the calling process.
- `fs.constants.X_OK` - `path` can be executed by the calling process. This has no effect on Windows (will behave like `fs.constants.F_OK`).

The final argument, `callback`, is a callback function that is invoked with a possible error argument. If any of the accessibility checks fail, the error argument will be an `Error` object. The following example checks if the file `/etc/passwd` can be read and written by the current process.

```
fs.access('/etc/passwd', fs.constants.R_OK | fs.constants.W_OK, (err) => {
  console.log(err ? 'no access!' : 'can read/write');
});
```

Using `fs.access()` to check for the accessibility of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file is not accessible.

write (NOT RECOMMENDED)

```
fs.access('myfile', (err) => {
  if (!err) {
    console.error('myfile already exists');
    return;
  }

  fs.open('myfile', 'wx', (err, fd) => {
    if (err) throw err;
    writeMyData(fd);
  });
});
```

write (RECOMMENDED)

```
fs.open('myfile', 'wx', (err, fd) => {
  if (err) {
    if (err.code === 'EEXIST') {
      console.error('myfile already exists');
      return;
    }

    throw err;
  }

  writeMyData(fd);
});
```

read (NOT RECOMMENDED)

```
fs.access('myfile', (err) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }

  fs.open('myfile', 'r', (err, fd) => {
    if (err) throw err;
    readMyData(fd);
  });
});
```

read (RECOMMENDED)

```
fs.open('myfile', 'r', (err, fd) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }

  readMyData(fd);
});
```

The "not recommended" examples above check for accessibility and then use the file; the "recommended" examples are better because they use the file directly and handle the error, if any.

In general, check for the accessibility of a file only if the file will not be used directly, for example when its accessibility is a signal from another process.

fs.accessSync(path[, mode])

▶ History

- path <string> | <Buffer> | <URL>
- mode <integer> **Default:** fs.constants.F_OK

Synchronously tests a user's permissions for the file or directory specified by path. The mode argument is an optional integer that specifies the accessibility checks to be performed. The following constants define the possible values of mode. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. fs.constants.W_OK | fs.constants.R_OK).

- fs.constants.F_OK - path is visible to the calling process. This is useful for determining if a file exists, but says nothing about rwx permissions. Default if no mode is specified.
- fs.constants.R_OK - path can be read by the calling process.
- fs.constants.W_OK - path can be written by the calling process.
- fs.constants.X_OK - path can be executed by the calling process. This has no effect on Windows (will behave like fs.constants.F_OK).

If any of the accessibility checks fail, an Error will be thrown. Otherwise, the method will return undefined.

```
try {
  fs.accessSync('etc/passwd', fs.constants.R_OK | fs.constants.W_OK);
  console.log('can read/write');
} catch (err) {
  console.error('no access!');
}
```

fs.appendFile(file, data[, options], callback)

▸ History

- file <string> | <Buffer> | <URL> | <number> filename or file descriptor
- data <string> | <Buffer>
- options <Object> | <string>
  - encoding <string> | <null> **Default:** 'utf8'
  - mode <integer> **Default:** 0o666
  - flag <string> **Default:** 'a'
- callback <Function>
  - err <Error>

Asynchronously append data to a file, creating the file if it does not yet exist. data can be a string or a Buffer.

Example:

```
fs.appendFile('message.txt', 'data to append', (err) => {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```

If options is a string, then it specifies the encoding. Example:

```
fs.appendFile('message.txt', 'data to append', 'utf8', callback);
```

The file may be specified as a numeric file descriptor that has been opened for appending (using fs.open() or fs.openSync()). The file descriptor will not be closed automatically.

```
fs.open('message.txt', 'a', (err, fd) => {
  if (err) throw err;
  fs.appendFile(fd, 'data to append', 'utf8', (err) => {
    fs.close(fd, (err) => {
      if (err) throw err;
    });
    if (err) throw err;
  });
});
```

fs.appendFileSync(file, data[, options])

▶ History

- `file` <string> | <Buffer> | <URL> | <number> filename or file descriptor
- `data` <string> | <Buffer>
- `options` <Object> | <string>
  - `encoding` <string> | <null> **Default:** `'utf8'`
  - `mode` <integer> **Default:** `0o666`
  - `flag` <string> **Default:** `'a'`

Synchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a Buffer .

Example:

```
try {
  fs.appendFileSync('message.txt', 'data to append');
  console.log('The "data to append" was appended to file!');
} catch (err) {
  /* Handle the error */
}
```

If `options` is a string, then it specifies the encoding. Example:

```
fs.appendFileSync('message.txt', 'data to append', 'utf8');
```

The `file` may be specified as a numeric file descriptor that has been opened for appending (using `fs.open()` or `fs.openSync()`). The file descriptor will not be closed automatically.

```
let fd;

try {
  fd = fs.openSync('message.txt', 'a');
  fs.appendFileSync(fd, 'data to append', 'utf8');
} catch (err) {
  /* Handle the error */
} finally {
  if (fd !== undefined)
    fs.closeSync(fd);
}
```

fs.chmod(path, mode, callback)

▶ History

* `path` <string> | <Buffer> | <URL>
* `mode` <integer>
* `callback` <Function>
  * `err` <Error>

Asynchronously changes the permissions of a file. No arguments other than a possible exception are given to the completion callback.

See also: chmod(2)

File modes

The `mode` argument used in both the `fs.chmod()` and `fs.chmodSync()` methods is a numeric bitmask created using a logical OR of the following constants:

Constant	Octal	Description
`fs.constants.S_IRUSR`	`0o400`	read by owner
`fs.constants.S_IWUSR`	`0o200`	write by owner
`fs.constants.S_IXUSR`	`0o100`	execute/search by owner
`fs.constants.S_IRGRP`	`0o40`	read by group
`fs.constants.S_IWGRP`	`0o20`	write by group
`fs.constants.S_IXGRP`	`0o10`	execute/search by group
`fs.constants.S_IROTH`	`0o4`	read by others
`fs.constants.S_IWOTH`	`0o2`	write by others
`fs.constants.S_IXOTH`	`0o1`	execute/search by others

An easier method of constructing the `mode` is to use a sequence of three octal digits (e.g. `765`). The left-most digit (`7` in the example), specifies the permissions for the file owner. The middle digit (`6` in the example), specifies permissions for the group. The right-most digit (`5` in the example), specifies the permissions for others.

Number	Description
`7`	read, write, and execute
`6`	read and write
`5`	read and execute
`4`	read only
`3`	write and execute
`2`	write only
`1`	execute only
`0`	no permission

For example, the octal value `0o765` means:

- The owner may read, write and execute the file.
- The group may read and write the file.
- Others may read and execute the file.

fs.chmodSync(path, mode)

▸ History

- `path` <string> | <Buffer> | <URL>
- `mode` <integer>

Synchronously changes the permissions of a file. Returns `undefined`. This is the synchronous version of fs.chmod() .

See also: chmod(2)

fs.chown(path, uid, gid, callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `uid` <integer>
- `gid` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronously changes owner and group of a file. No arguments other than a possible exception are given to the completion callback.

See also: chown(2)

fs.chownSync(path, uid, gid)

▸ History

- `path` <string> | <Buffer> | <URL>
- `uid` <integer>
- `gid` <integer>

Synchronously changes owner and group of a file. Returns `undefined`. This is the synchronous version of fs.chown() .

See also: chown(2)

fs.close(fd, callback)

▸ History

- `fd` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronous close(2) . No arguments other than a possible exception are given to the completion callback.

fs.closeSync(fd)

Added in: v0.1.21

- `fd` <integer>

Synchronous close(2) . Returns `undefined`.

fs.constants

Returns an object containing commonly used constants for file system operations. The specific constants currently defined are described in FS Constants .

fs.copyFile(src, dest[, flags], callback)

Added in: v8.5.0

- `src` <string> | <Buffer> | <URL> source filename to copy
- `dest` <string> | <Buffer> | <URL> destination filename of the copy operation
- `flags` <number> modifiers for copy operation. **Default:** `0`
- `callback` <Function>

Asynchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists. No arguments other than a possible exception are given to the callback function. Node.js makes no guarantees about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, Node.js will attempt to remove the destination.

`flags` is an optional integer that specifies the behavior of the copy operation. The only supported flag is `fs.constants.COPYFILE_EXCL`, which causes the copy operation to fail if `dest` already exists.

Example:

```
const fs = require('fs');

// destination.txt will be created or overwritten by default.
fs.copyFile('source.txt', 'destination.txt', (err) => {
  if (err) throw err;
  console.log('source.txt was copied to destination.txt');
});
```

If the third argument is a number, then it specifies `flags`, as shown in the following example.

```
const fs = require('fs');
const { COPYFILE_EXCL } = fs.constants;

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
fs.copyFile('source.txt', 'destination.txt', COPYFILE_EXCL, callback);
```

fs.copyFileSync(src, dest[, flags])

Added in: v8.5.0

- `src` <string> | <Buffer> | <URL> source filename to copy
- `dest` <string> | <Buffer> | <URL> destination filename of the copy operation
- `flags` <number> modifiers for copy operation. **Default:** `0`

Synchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists. Returns `undefined`. Node.js makes no guarantees about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, Node.js will attempt to remove the destination.

`flags` is an optional integer that specifies the behavior of the copy operation. The only supported flag is `fs.constants.COPYFILE_EXCL`, which causes the copy operation to fail if `dest` already exists.

Example:

```
const fs = require('fs');

// destination.txt will be created or overwritten by default.
fs.copyFileSync('source.txt', 'destination.txt');
console.log('source.txt was copied to destination.txt');
```

If the third argument is a number, then it specifies `flags`, as shown in the following example.

```
const fs = require('fs');
const { COPYFILE_EXCL } = fs.constants;

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
fs.copyFileSync('source.txt', 'destination.txt', COPYFILE_EXCL);
```

fs.createReadStream(path[, options])

▸ History

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `flags` <string>
  - `encoding` <string>
  - `fd` <integer>
  - `mode` <integer>
  - `autoClose` <boolean>
  - `start` <integer>
  - `end` <integer>
  - `highWaterMark` <integer>
- Returns: <ReadStream>

Returns a new `ReadStream` object. (See Readable Streams).

Be aware that, unlike the default value set for `highWaterMark` on a readable stream (16 kb), the stream returned by this method has a default value of 64 kb for the same parameter.

`options` is an object or string with the following defaults:

```
const defaults = {
  flags: 'r',
  encoding: null,
  fd: null,
  mode: 0o666,
  autoClose: true,
  highWaterMark: 64 * 1024
};
```

`options` can include `start` and `end` values to read a range of bytes from the file instead of the entire file. Both `start` and `end` are inclusive and start counting at 0. If `fd` is specified and `start` is omitted or `undefined`, `fs.createReadStream()` reads sequentially from the current file position. The `encoding` can be any one of those accepted by `Buffer`.

If `fd` is specified, `ReadStream` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. Note that `fd` should be blocking; non-blocking `fd`s should be passed to `net.Socket`.

If `autoClose` is false, then the file descriptor won't be closed, even if there's an error. It is the application's responsibility to close it and make

sure there's no file descriptor leak. If `autoClose` is set to true (default behavior), on `error` or `end` the file descriptor will be closed automatically.

`mode` sets the file mode (permission and sticky bits), but only if the file was created.

An example to read the last 10 bytes of a file which is 100 bytes long:

```
fs.createReadStream('sample.txt', { start: 90, end: 99 });
```

If `options` is a string, then it specifies the encoding.

fs.createWriteStream(path[, options])

▸ History

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `flags` <string>
  - `encoding` <string>
  - `fd` <integer>
  - `mode` <integer>
  - `autoClose` <boolean>
  - `start` <integer>
- Returns: <WriteStream>

Returns a new `WriteStream` object. (See Writable Stream).

`options` is an object or string with the following defaults:

```
const defaults = {
  flags: 'w',
  encoding: 'utf8',
  fd: null,
  mode: 0o666,
  autoClose: true
};
```

`options` may also include a `start` option to allow writing data at some position past the beginning of the file. Modifying a file rather than replacing it may require a `flags` mode of `r+` rather than the default mode `w`. The `encoding` can be any one of those accepted by `Buffer`.

If `autoClose` is set to true (default behavior) on `error` or `end` the file descriptor will be closed automatically. If `autoClose` is false, then the file descriptor won't be closed, even if there's an error. It is the application's responsibility to close it and make sure there's no file descriptor leak.

Like `ReadStream`, if `fd` is specified, `WriteStream` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. Note that `fd` should be blocking; non-blocking `fd`s should be passed to `net.Socket`.

If `options` is a string, then it specifies the encoding.

fs.exists(path, callback)

▸ History

Stability: 0 - Deprecated: Use `fs.stat()` or `fs.access()` instead.

- `path` <string> | <Buffer> | <URL>

- `callback` <Function>
  - `exists` <boolean>

Test whether or not the given path exists by checking with the file system. Then call the `callback` argument with either true or false. Example:

```
fs.exists('/etc/passwd', (exists) => {
  console.log(exists ? 'it\'s there' : 'no passwd!');
});
```

Note that the parameter to this callback is not consistent with other Node.js callbacks. Normally, the first parameter to a Node.js callback is an `err` parameter, optionally followed by other parameters. The `fs.exists()` callback has only one boolean parameter. This is one reason `fs.access()` is recommended instead of `fs.exists()`.

Using `fs.exists()` to check for the existence of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file does not exist.

write (NOT RECOMMENDED)

```
fs.exists('myfile', (exists) => {
  if (exists) {
    console.error('myfile already exists');
  } else {
    fs.open('myfile', 'wx', (err, fd) => {
      if (err) throw err;
      writeMyData(fd);
    });
  }
});
```

write (RECOMMENDED)

```
fs.open('myfile', 'wx', (err, fd) => {
  if (err) {
    if (err.code === 'EEXIST') {
      console.error('myfile already exists');
      return;
    }

    throw err;
  }

  writeMyData(fd);
});
```

read (NOT RECOMMENDED)

```
fs.exists('myfile', (exists) => {
  if (exists) {
    fs.open('myfile', 'r', (err, fd) => {
      if (err) throw err;
      readMyData(fd);
    });
  } else {
    console.error('myfile does not exist');
  }
});
```

read (RECOMMENDED)

```
fs.open('myfile', 'r', (err, fd) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }

  readMyData(fd);
});
```

The "not recommended" examples above check for existence and then use the file; the "recommended" examples are better because they use the file directly and handle the error, if any.

In general, check for the existence of a file only if the file won't be used directly, for example when its existence is a signal from another process.

fs.existsSync(path)

▸ History

- `path` <string> | <Buffer> | <URL>
- Returns: <boolean>

Synchronous version of `fs.exists()` . Returns `true` if the path exists, `false` otherwise.

Note that `fs.exists()` is deprecated, but `fs.existsSync()` is not. (The `callback` parameter to `fs.exists()` accepts parameters that are inconsistent with other Node.js callbacks. `fs.existsSync()` does not use a callback.)

fs.fchmod(fd, mode, callback)

▸ History

- `fd` <integer>
- `mode` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronous fchmod(2) . No arguments other than a possible exception are given to the completion callback.

fs.fchmodSync(fd, mode)

Added in: v0.4.7

- `fd` <integer>
- `mode` <integer>

Synchronous fchmod(2) . Returns `undefined`.

fs.fchown(fd, uid, gid, callback)

▸ History

- `fd` <integer>
- `uid` <integer>
- `gid` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronous fchown(2) . No arguments other than a possible exception are given to the completion callback.

fs.fchownSync(fd, uid, gid)

Added in: v0.4.7

- `fd` <integer>
- `uid` <integer>
- `gid` <integer>

Synchronous fchown(2) . Returns `undefined`.

fs.fdatasync(fd, callback)

▸ History

- `fd` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronous fdatasync(2) . No arguments other than a possible exception are given to the completion callback.

fs.fdatasyncSync(fd)

Added in: v0.1.96

- `fd` <integer>

Synchronous fdatasync(2) . Returns `undefined`.

fs.fstat(fd, callback)

▸ History

- `fd` <integer>
- `callback` <Function>
  - `err` <Error>
  - `stats` <fs.Stats>

Asynchronous fstat(2) . The callback gets two arguments `(err, stats)` where `stats` is an fs.Stats object. `fstat()` is identical to stat() , except that the file to be stat-ed is specified by the file descriptor `fd`.

fs.fstatSync(fd)

Added in: v0.1.95

- `fd` <integer>
- Returns: <fs.Stats>

Synchronous fstat(2) . Returns an instance of fs.Stats .

fs.fsync(fd, callback)

▸ History

- `fd` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronous fsync(2) . No arguments other than a possible exception are given to the completion callback.

fs.fsyncSync(fd)

Added in: v0.1.96

- `fd` <integer>

Synchronous fsync(2) . Returns `undefined`.

fs.ftruncate(fd[, len], callback)

▸ History

- `fd` <integer>
- `len` <integer> **Default:** `0`
- `callback` <Function>
  - `err` <Error>

Asynchronous ftruncate(2) . No arguments other than a possible exception are given to the completion callback.

If the file referred to by the file descriptor was larger than `len` bytes, only the first `len` bytes will be retained in the file.

For example, the following program retains only the first four bytes of the file

```
console.log(fs.readFileSync('temp.txt', 'utf8'));
// Prints: Node.js

// get the file descriptor of the file to be truncated
const fd = fs.openSync('temp.txt', 'r+');

// truncate the file to first four bytes
fs.ftruncate(fd, 4, (err) => {
  assert.ifError(err);
  console.log(fs.readFileSync('temp.txt', 'utf8'));
});
// Prints: Node
```

If the file previously was shorter than `len` bytes, it is extended, and the extended part is filled with null bytes ('\0'). For example,

```
console.log(fs.readFileSync('temp.txt', 'utf8'));
// Prints: Node.js

// get the file descriptor of the file to be truncated
const fd = fs.openSync('temp.txt', 'r+');

// truncate the file to 10 bytes, whereas the actual size is 7 bytes
fs.ftruncate(fd, 10, (err) => {
  assert.ifError(err);
  console.log(fs.readFileSync('temp.txt'));
});
// Prints: <Buffer 4e 6f 64 65 2e 6a 73 00 00 00>
// ('Node.js\0\0\0' in UTF8)
```

The last three bytes are null bytes ('\0'), to compensate the over-truncation.

fs.ftruncateSync(fd[, len])

Added in: v0.8.6

- `fd` <integer>
- `len` <integer> **Default:** `0`

Synchronous ftruncate(2). Returns `undefined`.

fs.futimes(fd, atime, mtime, callback)

▸ History

- `fd` <integer>
- `atime` <number> | <string> | <Date>
- `mtime` <number> | <string> | <Date>
- `callback` <Function>
  - `err` <Error>

Change the file system timestamps of the object referenced by the supplied file descriptor. See fs.utimes().

This function does not work on AIX versions before 7.1, it will return the error `UV_ENOSYS`.

fs.futimesSync(fd, atime, mtime)

▸ History

- `fd` <integer>
- `atime` <integer>
- `mtime` <integer>

Synchronous version of fs.futimes(). Returns `undefined`.

fs.lchmod(path, mode, callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `mode` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronous lchmod(2) . No arguments other than a possible exception are given to the completion callback.

Only available on macOS.

fs.lchmodSync(path, mode)

Deprecated since: v0.4.7

- `path` <string> | <Buffer> | <URL>
- `mode` <integer>

Synchronous lchmod(2) . Returns `undefined`.

fs.lchown(path, uid, gid, callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `uid` <integer>
- `gid` <integer>
- `callback` <Function>
  - `err` <Error>

Asynchronous lchown(2) . No arguments other than a possible exception are given to the completion callback.

fs.lchownSync(path, uid, gid)

Deprecated since: v0.4.7

- `path` <string> | <Buffer> | <URL>
- `uid` <integer>
- `gid` <integer>

Synchronous lchown(2) . Returns `undefined`.

fs.link(existingPath, newPath, callback)

▸ History

- `existingPath` <string> | <Buffer> | <URL>
- `newPath` <string> | <Buffer> | <URL>
- `callback` <Function>
  - `err` <Error>

Asynchronous link(2) . No arguments other than a possible exception are given to the completion callback.

fs.linkSync(existingPath, newPath)

▸ History

- `existingPath` <string> | <Buffer> | <URL>
- `newPath` <string> | <Buffer> | <URL>

Synchronous link(2) . Returns `undefined`.

fs.lstat(path, callback)

▸ History

- `path` <string> | <Buffer> | <URL>

- callback <Function>
  - err <Error>
  - stats <fs.Stats>

Asynchronous lstat(2). The callback gets two arguments (err, stats) where stats is a fs.Stats object. lstat() is identical to stat(), except that if path is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fs.lstatSync(path)

▸ History

- path <string> | <Buffer> | <URL>
- Returns: <fs.Stats>

Synchronous lstat(2). Returns an instance of fs.Stats.

fs.mkdir(path[, mode], callback)

▸ History

- path <string> | <Buffer> | <URL>
- mode <integer> **Default:** 0o777
- callback <Function>
  - err <Error>

Asynchronously creates a directory. No arguments other than a possible exception are given to the completion callback. mode defaults to 0o777.

See also: mkdir(2)

fs.mkdirSync(path[, mode])

▸ History

- path <string> | <Buffer> | <URL>
- mode <integer> **Default:** 0o777

Synchronously creates a directory. Returns undefined. This is the synchronous version of fs.mkdir().

See also: mkdir(2)

fs.mkdtemp(prefix[, options], callback)

▸ History

- prefix <string>
- options <string> | <Object>
  - encoding <string> **Default:** 'utf8'
- callback <Function>
  - err <Error>
  - folder <string>

Creates a unique temporary directory.

Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

The created folder path is passed as a string to the callback's second parameter.

The optional options argument can be a string specifying an encoding, or an object with an encoding property specifying the character encoding to use.

Example:

```
fs.mkdtemp(path.join(os.tmpdir(), 'foo-'), (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Prints: /tmp/foo-itXde2 or C:\Users\...\AppData\Local\Temp\foo-itXde2
});
```

The `fs.mkdtemp()` method will append the six randomly selected characters directly to the `prefix` string. For instance, given a directory `/tmp`, if the intention is to create a temporary directory *within* `/tmp`, the `prefix` *must* end with a trailing platform-specific path separator (`require('path').sep`).

```
// The parent directory for the new temporary directory
const tmpDir = os.tmpdir();

// This method is *INCORRECT*:
fs.mkdtemp(tmpDir, (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Will print something similar to `/tmpabc123`.
  // Note that a new temporary directory is created
  // at the file system root rather than *within*
  // the /tmp directory.
});

// This method is *CORRECT*:
const { sep } = require('path');
fs.mkdtemp(`${tmpDir}${sep}`, (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Will print something similar to `/tmp/abc123`.
  // A new temporary directory is created within
  // the /tmp directory.
});
```

fs.mkdtempSync(prefix[, options])

Added in: v5.10.0

- `prefix` <string>
- `options` <string> | <Object>
  - `encoding` <string> **Default:** `'utf8'`
- Returns: <string>

The synchronous version of `fs.mkdtemp()`. Returns the created folder path.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use.

fs.open(path, flags[, mode], callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `flags` <string> | <number>

- `mode` <integer> **Default:** `0o666`
- `callback` <Function>
  - `err` <Error>
  - `fd` <integer>

Asynchronous file open. See open(2) . `flags` can be:

- `'r'` - Open file for reading. An exception occurs if the file does not exist.
- `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs+'` - Open file for reading and writing in synchronous mode. Instructs the operating system to bypass the local file system cache.

  This is primarily useful for opening files on NFS mounts as it allows skipping the potentially stale local cache. It has a very real impact on I/O performance so using this flag is not recommended unless it is needed.

  Note that this doesn't turn `fs.open()` into a synchronous blocking call. If synchronous operation is desired `fs.openSync()` should be used.
- `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` - Like `'w'` but fails if `path` exists.
- `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` - Like `'w+'` but fails if `path` exists.
- `'a'` - Open file for appending. The file is created if it does not exist.
- `'ax'` - Like `'a'` but fails if `path` exists.
- `'as'` - Open file for appending in synchronous mode. The file is created if it does not exist.
- `'a+'` - Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` - Like `'a+'` but fails if `path` exists.
- `'as+'` - Open file for reading and appending in synchronous mode. The file is created if it does not exist.

`mode` sets the file mode (permission and sticky bits), but only if the file was created. It defaults to `0o666` (readable and writable).

The callback gets two arguments `(err, fd)` .

The exclusive flag `'x'` (`O_EXCL` flag in open(2)) ensures that `path` is newly created. On POSIX systems, `path` is considered to exist even if it is a symlink to a non-existent file. The exclusive flag may or may not work with network file systems.

`flags` can also be a number as documented by open(2) ; commonly used constants are available from `fs.constants` . On Windows, flags are translated to their equivalent ones where applicable, e.g. `O_WRONLY` to `FILE_GENERIC_WRITE` , or `O_EXCL|O_CREAT` to `CREATE_NEW` , as accepted by CreateFileW.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

The behavior of `fs.open()` is platform-specific for some flags. As such, opening a directory on macOS and Linux with the `'a+'` flag - see example below - will return an error. In contrast, on Windows and FreeBSD, a file descriptor will be returned.

```
// macOS and Linux
fs.open('<directory>', 'a+', (err, fd) => {
  // => [Error: EISDIR: illegal operation on a directory, open <directory>]
});

// Windows and FreeBSD
fs.open('<directory>', 'a+', (err, fd) => {
  // => null, <fd>
});
```

Some characters (`< > : " / \ | ? *`) are reserved under Windows as documented by Naming Files, Paths, and Namespaces. Under NTFS, if the filename contains a colon, Node.js will open a file system stream, as described by this MSDN page.

Functions based on `fs.open()` exhibit this behavior as well. eg. `fs.writeFile()`, `fs.readFile()`, etc.

Note: On Windows, opening an existing hidden file using the `w` flag (either through `fs.open()` or `fs.writeFile()`) will fail with `EPERM`. Existing hidden files can be opened for writing with the `r+` flag. A call to `fs.ftruncate()` can be used to reset the file contents.

fs.openSync(path, flags[, mode])

▸ History

- `path` <string> | <Buffer> | <URL>
- `flags` <string> | <number>
- `mode` <integer> **Default:** `0o666`
- Returns: <number>

Synchronous version of `fs.open()`. Returns an integer representing the file descriptor.

fs.read(fd, buffer, offset, length, position, callback)

▸ History

- `fd` <integer>
- `buffer` <Buffer> | <Uint8Array>
- `offset` <integer>
- `length` <integer>
- `position` <integer>
- `callback` <Function>
  - `err` <Error>
  - `bytesRead` <integer>
  - `buffer` <Buffer>

Read data from the file specified by `fd`.

`buffer` is the buffer that the data will be written to.

`offset` is the offset in the buffer to start writing at.

`length` is an integer specifying the number of bytes to read.

`position` is an argument specifying where to begin reading from in the file. If `position` is `null`, data will be read from the current file position, and the file position will be updated. If `position` is an integer, the file position will remain unchanged.

The callback is given the three arguments, `(err, bytesRead, buffer)`.

If this method is invoked as its `util.promisify()` ed version, it returns a Promise for an object with `bytesRead` and `buffer` properties.

fs.readdir(path[, options], callback)

▸ History

- path <string> | <Buffer> | <URL>
- options <string> | <Object>
  - encoding <string> **Default:** 'utf8'
- callback <Function>
  - err <Error>
  - files <string[]> | <Buffer[]>

Asynchronous readdir(3). Reads the contents of a directory. The callback gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.

The optional options argument can be a string specifying an encoding, or an object with an encoding property specifying the character encoding to use for the filenames passed to the callback. If the encoding is set to 'buffer', the filenames returned will be passed as Buffer objects.

fs.readdirSync(path[, options])

▸ History

- path <string> | <Buffer> | <URL>
- options <string> | <Object>
  - encoding <string> **Default:** 'utf8'
- Returns: <Array> An array of filenames

Synchronous readdir(3). Returns an array of filenames excluding '.' and '..'.

The optional options argument can be a string specifying an encoding, or an object with an encoding property specifying the character encoding to use for the filenames passed to the callback. If the encoding is set to 'buffer', the filenames returned will be passed as Buffer objects.

fs.readFile(path[, options], callback)

▸ History

- path <string> | <Buffer> | <URL> | <integer> filename or file descriptor
- options <Object> | <string>
  - encoding <string> | <null> **Default:** null
  - flag <string> **Default:** 'r'
- callback <Function>
  - err <Error>
  - data <string> | <Buffer>

Asynchronously reads the entire contents of a file. Example:

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments (err, data), where data is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

If options is a string, then it specifies the encoding. Example:

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

When the path is a directory, the behavior of `fs.readFile()` and `fs.readFileSync()` is platform-specific. On macOS, Linux, and Windows, an error will be returned. On FreeBSD, a representation of the directory's contents will be returned.

```
// macOS, Linux, and Windows
fs.readFile('<directory>', (err, data) => {
  // => [Error: EISDIR: illegal operation on a directory, read <directory>]
});

//  FreeBSD
fs.readFile('<directory>', (err, data) => {
  // => null, <data>
});
```

Any specified file descriptor has to support reading.

If a file descriptor is specified as the `path`, it will not be closed automatically.

The `fs.readFile()` function reads the entire file in a single threadpool request. To minimize threadpool task length variation, prefer the partitioned APIs `fs.read()` and `fs.createReadStream()` when reading files as part of fulfilling a client request.

fs.readFileSync(path[, options])

▸ History

- `path` <string> | <Buffer> | <URL> | <integer> filename or file descriptor
- `options` <Object> | <string>
  - `encoding` <string> | <null> **Default:** `null`
  - `flag` <string> **Default:** `'r'`
- Returns: <string> | <Buffer>

Synchronous version of `fs.readFile()`. Returns the contents of the `path`.

If the `encoding` option is specified then this function returns a string. Otherwise it returns a buffer.

Similar to `fs.readFile()`, when the path is a directory, the behavior of `fs.readFileSync()` is platform-specific.

```
// macOS, Linux, and Windows
fs.readFileSync('<directory>');
// => [Error: EISDIR: illegal operation on a directory, read <directory>]

//  FreeBSD
fs.readFileSync('<directory>'); // => null, <data>
```

fs.readlink(path[, options], callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `encoding` <string> **Default:** `'utf8'`
- `callback` <Function>
  - `err` <Error>
  - `linkString` <string> | <Buffer>

Asynchronous readlink(2). The callback gets two arguments `(err, linkString)`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path passed to the callback. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `Buffer` object.

fs.readlinkSync(path[, options])

▸ History

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `encoding` <string> **Default:** `'utf8'`
- Returns: <string> | <Buffer>

Synchronous readlink(2). Returns the symbolic link's string value.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path passed to the callback. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `Buffer` object.

fs.readSync(fd, buffer, offset, length, position)

▸ History

- `fd` <integer>
- `buffer` <Buffer> | <Uint8Array>
- `offset` <integer>
- `length` <integer>
- `position` <integer>
- Returns: <number>

Synchronous version of fs.read(). Returns the number of `bytesRead`.

fs.realpath(path[, options], callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `encoding` <string> **Default:** `'utf8'`
- `callback` <Function>
  - `err` <Error>
  - `resolvedPath` <string> | <Buffer>

Asynchronously computes the canonical pathname by resolving `.`, `..` and symbolic links.

Note that "canonical" does not mean "unique": hard links and bind mounts can expose a file system entity through many pathnames.

This function behaves like realpath(3), with some exceptions:

1. No case conversion is performed on case-insensitive file systems.
2. The maximum number of symbolic links is platform-independent and generally (much) higher than what the native realpath(3) implementation supports.

The `callback` gets two arguments `(err, resolvedPath)`. May use `process.cwd` to resolve relative paths.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character

encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

If `path` resolves to a socket or a pipe, the function will return a system dependent name for that object.

fs.realpath.native(path[, options], callback)

Added in: v9.2.0

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `encoding` <string> **Default:** `'utf8'`
- `callback` <Function>
  - `err` <Error>
  - `resolvedPath` <string> | <Buffer>

Asynchronous realpath(3).

The `callback` gets two arguments `(err, resolvedPath)`.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

fs.realpathSync(path[, options])

▸ History

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `encoding` <string> **Default:** `'utf8'`
- Returns: <string> | <Buffer>

Synchronously computes the canonical pathname by resolving `.`, `..` and symbolic links.

Note that "canonical" does not mean "unique": hard links and bind mounts can expose a file system entity through many pathnames.

This function behaves like realpath(3), with some exceptions:

1. No case conversion is performed on case-insensitive file systems.
2. The maximum number of symbolic links is platform-independent and generally (much) higher than what the native realpath(3) implementation supports.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the returned value. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

If `path` resolves to a socket or a pipe, the function will return a system dependent name for that object.

fs.realpathSync.native(path[, options])

Added in: v9.2.0

- `path` <string> | <Buffer> | <URL>
- `options` <string> | <Object>
  - `encoding` <string> **Default:** `'utf8'`
- Returns: <string> | <Buffer>

Synchronous realpath(3).

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

fs.rename(oldPath, newPath, callback)

▶ History

- `oldPath` <string> | <Buffer> | <URL>
- `newPath` <string> | <Buffer> | <URL>
- `callback` <Function>
  - `err` <Error>

Asynchronously rename file at `oldPath` to the pathname provided as `newPath`. In the case that `newPath` already exists, it will be overwritten. No arguments other than a possible exception are given to the completion callback.

See also: rename(2).

```
fs.rename('oldFile.txt', 'newFile.txt', (err) => {
  if (err) throw err;
  console.log('Rename complete!');
});
```

fs.renameSync(oldPath, newPath)

▶ History

- `oldPath` <string> | <Buffer> | <URL>
- `newPath` <string> | <Buffer> | <URL>

Synchronous rename(2). Returns `undefined`.

fs.rmdir(path, callback)

▶ History

- `path` <string> | <Buffer> | <URL>
- `callback` <Function>
  - `err` <Error>

Asynchronous rmdir(2). No arguments other than a possible exception are given to the completion callback.

Using `fs.rmdir()` on a file (not a directory) results in an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

fs.rmdirSync(path)

▶ History

- `path` <string> | <Buffer> | <URL>

Synchronous rmdir(2). Returns `undefined`.

Using `fs.rmdirSync()` on a file (not a directory) results in an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

fs.stat(path, callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `callback` <Function>
  - `err` <Error>
  - `stats` <fs.Stats>

Asynchronous stat(2). The callback gets two arguments `(err, stats)` where `stats` is an fs.Stats object.

In case of an error, the `err.code` will be one of Common System Errors.

Using `fs.stat()` to check for the existence of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Instead, user code should open/read/write the file directly and handle the error raised if the file is not available.

To check if a file exists without manipulating it afterwards, fs.access() is recommended.

fs.statSync(path)

▸ History

- `path` <string> | <Buffer> | <URL>
- Returns: <fs.Stats>

Synchronous stat(2). Returns an instance of fs.Stats.

fs.symlink(target, path[, type], callback)

▸ History

- `target` <string> | <Buffer> | <URL>
- `path` <string> | <Buffer> | <URL>
- `type` <string> **Default:** `'file'`
- `callback` <Function>
  - `err` <Error>

Asynchronous symlink(2). No arguments other than a possible exception are given to the completion callback. The `type` argument can be set to `'dir'`, `'file'`, or `'junction'` (default is `'file'`) and is only available on Windows (ignored on other platforms). Note that Windows junction points require the destination path to be absolute. When using `'junction'`, the `target` argument will automatically be normalized to absolute path.

Here is an example below:

```
fs.symlink('./foo', './new-port', callback);
```

It creates a symbolic link named "new-port" that points to "foo".

fs.symlinkSync(target, path[, type])

▸ History

- `target` <string> | <Buffer> | <URL>
- `path` <string> | <Buffer> | <URL>
- `type` <string> **Default:** `'file'`

Synchronous symlink(2). Returns `undefined`.

fs.truncate(path[, len], callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `len` <integer> **Default:** `0`
- `callback` <Function>
  - `err` <Error>

Asynchronous truncate(2). No arguments other than a possible exception are given to the completion callback. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncate()` is called.

Passing a file descriptor is deprecated and may result in an error being thrown in the future.

fs.truncateSync(path[, len])

Added in: v0.8.6

- `path` <string> | <Buffer> | <URL>
- `len` <integer> **Default:** `0`

Synchronous truncate(2). Returns `undefined`. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncateSync()` is called.

Passing a file descriptor is deprecated and may result in an error being thrown in the future.

fs.unlink(path, callback)

▸ History

- `path` <string> | <Buffer> | <URL>
- `callback` <Function>
  - `err` <Error>

Asynchronously removes a file or symbolic link. No arguments other than a possible exception are given to the completion callback.

```
// Assuming that 'path/file.txt' is a regular file.
fs.unlink('path/file.txt', (err) => {
  if (err) throw err;
  console.log('path/file.txt was deleted');
});
```

`fs.unlink()` will not work on a directory, empty or otherwise. To remove a directory, use fs.rmdir().

See also: unlink(2)

fs.unlinkSync(path)

▸ History

- `path` <string> | <Buffer> | <URL>

Synchronous unlink(2). Returns `undefined`.

fs.unwatchFile(filename[, listener])

Added in: v0.1.31

- `filename` <string> | <Buffer> | <URL>
- `listener` <Function> Optional, a listener previously attached using `fs.watchFile()`

Stop watching for changes on `filename`. If `listener` is specified, only that particular listener is removed. Otherwise, *all* listeners are removed, effectively stopping watching of `filename`.

Calling `fs.unwatchFile()` with a filename that is not being watched is a no-op, not an error.

Using fs.watch() is more efficient than fs.watchFile() and fs.unwatchFile(). fs.watch() should be used instead of fs.watchFile() and fs.unwatchFile() when possible.

fs.utimes(path, atime, mtime, callback)

▶ History

- path <string> | <Buffer> | <URL>
- atime <number> | <string> | <Date>
- mtime <number> | <string> | <Date>
- callback <Function>
  - err <Error>

Change the file system timestamps of the object referenced by `path`.

The `atime` and `mtime` arguments follow these rules:

- Values can be either numbers representing Unix epoch time, `Date` s, or a numeric string like `'123456789.0'`.
- If the value can not be converted to a number, or is `NaN`, `Infinity` or `-Infinity`, a `Error` will be thrown.

fs.utimesSync(path, atime, mtime)

▶ History

- path <string> | <Buffer> | <URL>
- atime <integer>
- mtime <integer>

Synchronous version of fs.utimes(). Returns `undefined`.

fs.watch(filename[, options][, listener])

▶ History

- filename <string> | <Buffer> | <URL>
- options <string> | <Object>
  - persistent <boolean> Indicates whether the process should continue to run as long as files are being watched. **Default:** `true`
  - recursive <boolean> Indicates whether all subdirectories should be watched, or only the current directory. This applies when a directory is specified, and only on supported platforms (See Caveats). **Default:** `false`
  - encoding <string> Specifies the character encoding to be used for the filename passed to the listener. **Default:** `'utf8'`
- listener <Function> | <undefined> **Default:** `undefined`
  - eventType <string>
  - filename <string> | <Buffer>

Watch for changes on `filename`, where `filename` is either a file or a directory. The returned object is a fs.FSWatcher.

The second argument is optional. If `options` is provided as a string, it specifies the `encoding`. Otherwise `options` should be passed as an object.

The listener callback gets two arguments `(eventType, filename)`. `eventType` is either `'rename'` or `'change'`, and `filename` is the name of the file which triggered the event.

Note that on most platforms, `'rename'` is emitted whenever a filename appears or disappears in the directory.

Also note the listener callback is attached to the `'change'` event fired by fs.FSWatcher, but it is not the same thing as the `'change'` value of `eventType`.

Caveats

The `fs.watch` API is not 100% consistent across platforms, and is unavailable in some situations.

The recursive option is only supported on macOS and Windows.

Availability

This feature depends on the underlying operating system providing a way to be notified of filesystem changes.

- On Linux systems, this uses inotify
- On BSD systems, this uses kqueue
- On macOS, this uses kqueue for files and FSEvents for directories.
- On SunOS systems (including Solaris and SmartOS), this uses event ports .
- On Windows systems, this feature depends on ReadDirectoryChangesW .
- On Aix systems, this feature depends on AHAFS , which must be enabled.

If the underlying functionality is not available for some reason, then `fs.watch` will not be able to function. For example, watching files or directories can be unreliable, and in some cases impossible, on network file systems (NFS, SMB, etc), or host file systems when using virtualization software such as Vagrant, Docker, etc.

It is still possible to use `fs.watchFile()`, which uses stat polling, but this method is slower and less reliable.

Inodes

On Linux and macOS systems, `fs.watch()` resolves the path to an inode and watches the inode. If the watched path is deleted and recreated, it is assigned a new inode. The watch will emit an event for the delete but will continue watching the *original* inode. Events for the new inode will not be emitted. This is expected behavior.

AIX files retain the same inode for the lifetime of a file. Saving and closing a watched file on AIX will result in two notifications (one for adding new content, and one for truncation).

Filename Argument

Providing `filename` argument in the callback is only supported on Linux, macOS, Windows, and AIX. Even on supported platforms, `filename` is not always guaranteed to be provided. Therefore, don't assume that `filename` argument is always provided in the callback, and have some fallback logic if it is null.

```
fs.watch('somedir', (eventType, filename) => {
  console.log(`event type is: ${eventType}`);
  if (filename) {
    console.log(`filename provided: ${filename}`);
  } else {
    console.log('filename not provided');
  }
});
```

fs.watchFile(filename[, options], listener)

▸ History

- `filename` <string> | <Buffer> | <URL>
- `options` <Object>
  - `persistent` <boolean> **Default:** `true`
  - `interval` <integer> **Default:** `5007`
- `listener` <Function>
  - `current` <fs.Stats>
  - `previous` <fs.Stats>

Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.

The `options` argument may be omitted. If provided, it should be an object. The `options` object may contain a boolean named `persistent` that indicates whether the process should continue to run as long as files are being watched. The `options` object may specify an `interval` property indicating how often the target should be polled in milliseconds. The default is `{ persistent: true, interval: 5007 }`.

The `listener` gets two arguments the current stat object and the previous stat object:

```
fs.watchFile('message.text', (curr, prev) => {
  console.log(`the current mtime is: ${curr.mtime}`);
  console.log(`the previous mtime was: ${prev.mtime}`);
});
```

These stat objects are instances of `fs.Stat`.

To be notified when the file was modified, not just accessed, it is necessary to compare `curr.mtime` and `prev.mtime`.

When an `fs.watchFile` operation results in an `ENOENT` error, it will invoke the listener once, with all the fields zeroed (or, for dates, the Unix Epoch). In Windows, `blksize` and `blocks` fields will be `undefined`, instead of zero. If the file is created later on, the listener will be called again, with the latest stat objects. This is a change in functionality since v0.10.

Using `fs.watch()` is more efficient than `fs.watchFile` and `fs.unwatchFile`. `fs.watch` should be used instead of `fs.watchFile` and `fs.unwatchFile` when possible.

When a file being watched by `fs.watchFile()` disappears and reappears, then the `previousStat` reported in the second callback event (the file's reappearance) will be the same as the `previousStat` of the first callback event (its disappearance).

This happens when:

- the file is deleted, followed by a restore
- the file is renamed twice - the second time back to its original name

fs.write(fd, buffer[, offset[, length[, position]]], callback)

▸ History

- `fd` <integer>
- `buffer` <Buffer> | <Uint8Array>
- `offset` <integer>
- `length` <integer>
- `position` <integer>
- `callback` <Function>
  - `err` <Error>
  - `bytesWritten` <integer>
  - `buffer` <Buffer> | <Uint8Array>

Write `buffer` to the file specified by `fd`.

`offset` determines the part of the buffer to be written, and `length` is an integer specifying the number of bytes to write.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'`, the data will be written at the current position. See pwrite(2).

The callback will be given three arguments `(err, bytesWritten, buffer)` where `bytesWritten` specifies how many *bytes* were written from `buffer`.

If this method is invoked as its `util.promisify()` ed version, it returns a Promise for an object with `bytesWritten` and `buffer` properties.

Note that it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends

the data to the end of the file.

fs.write(fd, string[, position[, encoding]], callback)

▸ History

- fd <integer>
- string <string>
- position <integer>
- encoding <string>
- callback <Function>
  - err <Error>
  - written <integer>
  - string <string>

Write `string` to the file specified by `fd`. If `string` is not a string, then the value will be coerced to one.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'` the data will be written at the current position. See pwrite(2).

`encoding` is the expected string encoding.

The callback will receive the arguments `(err, written, string)` where `written` specifies how many *bytes* the passed string required to be written. Note that bytes written is not the same as string characters. See Buffer.byteLength.

Unlike when writing `buffer`, the entire string must be written. No substring may be specified. This is because the byte offset of the resulting data may not be the same as the string offset.

Note that it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

fs.writeFile(file, data[, options], callback)

▸ History

- file <string> | <Buffer> | <URL> | <integer> filename or file descriptor
- data <string> | <Buffer> | <Uint8Array>
- options <Object> | <string>
  - encoding <string> | <null> **Default:** `'utf8'`
  - mode <integer> **Default:** `0o666`
  - flag <string> **Default:** `'w'`
- callback <Function>
  - err <Error>

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer.

The `encoding` option is ignored if `data` is a buffer. It defaults to `'utf8'`.

Example:

```
fs.writeFile('message.txt', 'Hello Node.js', (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

If `options` is a string, then it specifies the encoding. Example:

```
fs.writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

Any specified file descriptor has to support writing.

Note that it is unsafe to use `fs.writeFile` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

If a file descriptor is specified as the `file`, it will not be closed automatically.

fs.writeFileSync(file, data[, options])

▸ History

- `file` <string> | <Buffer> | <URL> | <integer> filename or file descriptor
- `data` <string> | <Buffer> | <Uint8Array>
- `options` <Object> | <string>
  - `encoding` <string> | <null> **Default:** `'utf8'`
  - `mode` <integer> **Default:** `0o666`
  - `flag` <string> **Default:** `'w'`

The synchronous version of fs.writeFile(). Returns `undefined`.

fs.writeSync(fd, buffer[, offset[, length[, position]]])

▸ History

- `fd` <integer>
- `buffer` <Buffer> | <Uint8Array>
- `offset` <integer>
- `length` <integer>
- `position` <integer>
- Returns: <number>

fs.writeSync(fd, string[, position[, encoding]])

▸ History

- `fd` <integer>
- `string` <string>
- `position` <integer>
- `encoding` <string>
- Returns: <number>

Synchronous versions of fs.write(). Returns the number of bytes written.

FS Constants

The following constants are exported by `fs.constants`.

Not every constant will be available on every operating system.

File Access Constants

The following constants are meant for use with fs.access() .

Constant	Description
F_OK	Flag indicating that the file is visible to the calling process.
R_OK	Flag indicating that the file can be read by the calling process.
W_OK	Flag indicating that the file can be written by the calling process.
X_OK	Flag indicating that the file can be executed by the calling process.

File Open Constants

The following constants are meant for use with fs.open() .

Constant	Description
O_RDONLY	Flag indicating to open a file for read-only access.
O_WRONLY	Flag indicating to open a file for write-only access.
O_RDWR	Flag indicating to open a file for read-write access.
O_CREAT	Flag indicating to create the file if it does not already exist.
O_EXCL	Flag indicating that opening a file should fail if the O_CREAT flag is set and the file already exists.
O_NOCTTY	Flag indicating that if path identifies a terminal device, opening the path shall not cause that terminal to become the controlling terminal for the process (if the process does not already have one).
O_TRUNC	Flag indicating that if the file exists and is a regular file, and the file is opened successfully for write access, its length shall be truncated to zero.
O_APPEND	Flag indicating that data will be appended to the end of the file.
O_DIRECTORY	Flag indicating that the open should fail if the path is not a directory.
O_NOATIME	Flag indicating reading accesses to the file system will no longer result in an update to the atime information associated with the file. This flag is available on Linux operating systems only.
O_NOFOLLOW	Flag indicating that the open should fail if the path is a symbolic link.
O_SYNC	Flag indicating that the file is opened for synchronized I/O with write operations waiting for file integrity.
O_DSYNC	Flag indicating that the file is opened for synchronized I/O with write operations waiting for data integrity.
O_SYMLINK	Flag indicating to open the symbolic link itself rather than the resource it is pointing to.
O_DIRECT	When set, an attempt will be made to minimize caching effects of file I/O.
O_NONBLOCK	Flag indicating to open the file in nonblocking mode when possible.

File Type Constants

The following constants are meant for use with the fs.Stats object's mode property for determining a file's type.

Constant	Description

S_IFMT	Bit mask used to extract the file type code.
S_IFREG	File type constant for a regular file.
S_IFDIR	File type constant for a directory.
S_IFCHR	File type constant for a character-oriented device file.
S_IFBLK	File type constant for a block-oriented device file.
S_IFIFO	File type constant for a FIFO/pipe.
S_IFLNK	File type constant for a symbolic link.
S_IFSOCK	File type constant for a socket.

File Mode Constants

The following constants are meant for use with the fs.Stats object's `mode` property for determining the access permissions for a file.

Constant	Description
S_IRWXU	File mode indicating readable, writable, and executable by owner.
S_IRUSR	File mode indicating readable by owner.
S_IWUSR	File mode indicating writable by owner.
S_IXUSR	File mode indicating executable by owner.
S_IRWXG	File mode indicating readable, writable, and executable by group.
S_IRGRP	File mode indicating readable by group.
S_IWGRP	File mode indicating writable by group.
S_IXGRP	File mode indicating executable by group.
S_IRWXO	File mode indicating readable, writable, and executable by others.
S_IROTH	File mode indicating readable by others.
S_IWOTH	File mode indicating writable by others.
S_IXOTH	File mode indicating executable by others.

Global Objects

These objects are available in all modules. The following variables may appear to be global but are not. They exist only in the scope of modules, see the module system documentation:

- __dirname
- __filename
- exports
- module

- require()

The objects listed here are specific to Node.js. There are a number of built-in objects that are part of the JavaScript language itself, which are also globally accessible.

Class: Buffer

Added in: v0.1.103

- <Function>

Used to handle binary data. See the buffer section.

__dirname

This variable may appear to be global but is not. See __dirname.

__filename

This variable may appear to be global but is not. See __filename.

clearImmediate(immediateObject)

Added in: v0.9.1

clearImmediate is described in the timers section.

clearInterval(intervalObject)

Added in: v0.0.1

clearInterval is described in the timers section.

clearTimeout(timeoutObject)

Added in: v0.0.1

clearTimeout is described in the timers section.

console

Added in: v0.1.100

- <Object>

Used to print to stdout and stderr. See the console section.

exports

This variable may appear to be global but is not. See exports.

global

Added in: v0.1.27

- <Object> The global namespace object.

In browsers, the top-level scope is the global scope. This means that within the browser `var something` will define a new global variable. In Node.js this is different. The top-level scope is not the global scope; `var something` inside a Node.js module will be local to that module.

module

This variable may appear to be global but is not. See module.

process

Added in: v0.1.7

- <Object>

The process object. See the process object section.

require()

This variable may appear to be global but is not. See require() .

setImmediate(callback[, ...args])

Added in: v0.9.1

setImmediate is described in the timers section.

setInterval(callback, delay[, ...args])

Added in: v0.0.1

setInterval is described in the timers section.

setTimeout(callback, delay[, ...args])

Added in: v0.0.1

setTimeout is described in the timers section.

HTTP

Stability: 2 - Stable

To use the HTTP server and client one must `require('http')` .

The HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses — the user is able to stream data.

HTTP message headers are represented by an object like this:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'host': 'mysite.com',
  'accept': '*/*' }
```

Keys are lowercased. Values are not modified.

In order to support the full spectrum of possible HTTP applications, Node.js's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.

See message.headers for details on how duplicate headers are handled.

The raw headers as they were received are retained in the `rawHeaders` property, which is an array of `[key, value, key2, value2, ...]` . For example, the previous message header object might have a `rawHeaders` list like the following:

```
[ 'ConTent-Length', '123456',
  'content-LENGTH', '123',
  'content-type', 'text/plain',
  'CONNECTION', 'keep-alive',
  'Host', 'mysite.com',
  'accepT', '*/*' ]
```

Class: http.Agent

Added in: v0.3.4

An `Agent` is responsible for managing connection persistence and reuse for HTTP clients. It maintains a queue of pending requests for a given host and port, reusing a single socket connection for each until the queue is empty, at which time the socket is either destroyed or put into a pool where it is kept to be used again for requests to the same host and port. Whether it is destroyed or pooled depends on the `keepAlive` option .

Pooled connections have TCP Keep-Alive enabled for them, but servers may still close idle connections, in which case they will be removed from the pool and a new connection will be made when a new HTTP request is made for that host and port. Servers may also refuse to allow multiple requests over the same connection, in which case the connection will have to be remade for every request and cannot be pooled. The `Agent` will still make the requests to that server, but each one will occur over a new connection.

When a connection is closed by the client or the server, it is removed from the pool. Any unused sockets in the pool will be unrefed so as not to keep the Node.js process running when there are no outstanding requests. (see socket.unref()).

It is good practice, to destroy() an `Agent` instance when it is no longer in use, because unused sockets consume OS resources.

Sockets are removed from an agent when the socket emits either a `'close'` event or an `'agentRemove'` event. When intending to keep one HTTP request open for a long time without keeping it in the agent, something like the following may be done:

```
http.get(options, (res) => {
  // Do stuff
}).on('socket', (socket) => {
  socket.emit('agentRemove');
});
```

An agent may also be used for an individual request. By providing `{agent: false}` as an option to the `http.get()` or `http.request()` functions, a one-time use `Agent` with default options will be used for the client connection.

`agent:false` :

```
http.get({
  hostname: 'localhost',
  port: 80,
  path: '/',
  agent: false  // create a new agent just for this one request
}, (res) => {
  // Do stuff with response
});
```

new Agent(options)

Added in: v0.3.4

- `options` <Object> Set of configurable options to set on the agent. Can have the following fields:
  - `keepAlive` <boolean> Keep sockets around even when there are no outstanding requests, so they can be used for future requests without having to reestablish a TCP connection. Defaults to `false`
  - `keepAliveMsecs` <number> When using the `keepAlive` option, specifies the initial delay for TCP Keep-Alive packets. Ignored when

the `keepAlive` option is `false` or `undefined`. Defaults to `1000`.

- `maxSockets` <number> Maximum number of sockets to allow per host. Defaults to `Infinity`.
- `maxFreeSockets` <number> Maximum number of sockets to leave open in a free state. Only relevant if `keepAlive` is set to `true`. **Default:** `256`.

The default http.globalAgent that is used by http.request() has all of these values set to their respective defaults.

To configure any of them, a custom http.Agent instance must be created.

```
const http = require('http');
const keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

agent.createConnection(options[, callback])

Added in: v0.11.4

- `options` <Object> Options containing connection details. Check net.createConnection() for the format of the options
- `callback` <Function> Callback function that receives the created socket
- Returns: <net.Socket>

Produces a socket/stream to be used for HTTP requests.

By default, this function is the same as net.createConnection(). However, custom agents may override this method in case greater flexibility is desired.

A socket/stream can be supplied in one of two ways: by returning the socket/stream from this function, or by passing the socket/stream to `callback`.

`callback` has a signature of `(err, stream)`.

agent.keepSocketAlive(socket)

Added in: v8.1.0

- `socket` <net.Socket>

Called when `socket` is detached from a request and could be persisted by the Agent. Default behavior is to:

```
socket.setKeepAlive(true, this.keepAliveMsecs);
socket.unref();
return true;
```

This method can be overridden by a particular `Agent` subclass. If this method returns a falsy value, the socket will be destroyed instead of persisting it for use with the next request.

agent.reuseSocket(socket, request)

Added in: v8.1.0

- `socket` <net.Socket>
- `request` <http.ClientRequest>

Called when `socket` is attached to `request` after being persisted because of the keep-alive options. Default behavior is to:

```
socket.ref();
```

This method can be overridden by a particular `Agent` subclass.

agent.destroy()

Added in: v0.11.4

Destroy any sockets that are currently in use by the agent.

It is usually not necessary to do this. However, if using an agent with `keepAlive` enabled, then it is best to explicitly shut down the agent when it will no longer be used. Otherwise, sockets may hang open for quite a long time before the server terminates them.

agent.freeSockets

Added in: v0.11.4

- <Object>

An object which contains arrays of sockets currently awaiting use by the agent when `keepAlive` is enabled. Do not modify.

agent.getName(options)

Added in: v0.11.4

- `options` <Object> A set of options providing information for name generation
  - `host` <string> A domain name or IP address of the server to issue the request to
  - `port` <number> Port of remote server
  - `localAddress` <string> Local interface to bind for network connections when issuing the request
  - `family` <integer> Must be 4 or 6 if this doesn't equal `undefined`.
- Returns: <string>

Get a unique name for a set of request options, to determine whether a connection can be reused. For an HTTP agent, this returns `host:port:localAddress` or `host:port:localAddress:family`. For an HTTPS agent, the name includes the CA, cert, ciphers, and other HTTPS/TLS-specific options that determine socket reusability.

agent.maxFreeSockets

Added in: v0.11.7

- <number>

By default set to 256. For agents with `keepAlive` enabled, this sets the maximum number of sockets that will be left open in the free state.

agent.maxSockets

Added in: v0.3.6

- <number>

By default set to Infinity. Determines how many concurrent sockets the agent can have open per origin. Origin is the returned value of `agent.getName()`.

agent.requests

Added in: v0.5.9

- <Object>

An object which contains queues of requests that have not yet been assigned to sockets. Do not modify.

agent.sockets

Added in: v0.3.6

- <Object>

An object which contains arrays of sockets currently in use by the agent. Do not modify.

Class: http.ClientRequest

Added in: v0.1.17

This object is created internally and returned from http.request() . It represents an *in-progress* request whose header has already been queued. The header is still mutable using the setHeader(name, value) , getHeader(name) , removeHeader(name) API. The actual header will be sent along with the first data chunk or when calling request.end() .

To get the response, add a listener for 'response' to the request object. 'response' will be emitted from the request object when the response headers have been received. The 'response' event is executed with one argument which is an instance of http.IncomingMessage .

During the 'response' event, one can add listeners to the response object; particularly to listen for the `'data'` event.

If no 'response' handler is added, then the response will be entirely discarded. However, if a 'response' event handler is added, then the data from the response object **must** be consumed, either by calling `response.read()` whenever there is a `'readable'` event, or by adding a `'data'` handler, or by calling the `.resume()` method. Until the data is consumed, the `'end'` event will not fire. Also, until the data is read it will consume memory that can eventually lead to a 'process out of memory' error.

Node.js does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

The request implements the Writable Stream interface. This is an EventEmitter with the following events:

Event: 'abort'

Added in: v1.4.1

Emitted when the request has been aborted by the client. This event is only emitted on the first call to `abort()`.

Event: 'connect'

Added in: v0.7.0

- `response` <http.IncomingMessage>
- `socket` <net.Socket>
- `head` <Buffer>

Emitted each time a server responds to a request with a `CONNECT` method. If this event is not being listened for, clients receiving a `CONNECT` method will have their connections closed.

A client and server pair demonstrating how to listen for the `'connect'` event:

```
const http = require('http');
const net = require('net');
const url = require('url');

// Create an HTTP tunneling proxy
const proxy = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('okay');
});
proxy.on('connect', (req, cltSocket, head) => {
  // connect to an origin server
  const srvUrl = url.parse(`http://${req.url}`);
  const srvSocket = net.connect(srvUrl.port, srvUrl.hostname, () => {
    cltSocket.write('HTTP/1.1 200 Connection Established\r\n' +
                    'Proxy-agent: Node.js-Proxy\r\n' +
                    '\r\n');
    srvSocket.write(head);
    srvSocket.pipe(cltSocket);
    cltSocket.pipe(srvSocket);
  });
});

// now that proxy is running
proxy.listen(1337, '127.0.0.1', () => {

  // make a request to a tunneling proxy
  const options = {
    port: 1337,
    hostname: '127.0.0.1',
    method: 'CONNECT',
    path: 'www.google.com:80'
  };

  const req = http.request(options);
  req.end();

  req.on('connect', (res, socket, head) => {
    console.log('got connected!');

    // make a request over an HTTP tunnel
    socket.write('GET / HTTP/1.1\r\n' +
                 'Host: www.google.com:80\r\n' +
                 'Connection: close\r\n' +
                 '\r\n');
    socket.on('data', (chunk) => {
      console.log(chunk.toString());
    });
    socket.on('end', () => {
      proxy.close();
    });
  });
});
```

Event: 'continue'

Added in: v0.3.2

Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.

Event: 'response'

Added in: v0.1.0

- `response` <http.IncomingMessage>

Emitted when a response is received to this request. This event is emitted only once.

Event: 'socket'

Added in: v0.5.3

- `socket` <net.Socket>

Emitted after a socket is assigned to this request.

Event: 'timeout'

Added in: v0.7.8

Emitted when the underlying socket times out from inactivity. This only notifies that the socket has been idle. The request must be aborted manually.

See also: request.setTimeout()

Event: 'upgrade'

Added in: v0.1.94

- `response` <http.IncomingMessage>
- `socket` <net.Socket>
- `head` <Buffer>

Emitted each time a server responds to a request with an upgrade. If this event is not being listened for, clients receiving an upgrade header will have their connections closed.

A client server pair demonstrating how to listen for the `'upgrade'` event.

```
const http = require('http');

// Create an HTTP server
const srv = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('okay');
});
srv.on('upgrade', (req, socket, head) => {
  socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n' +
               'Upgrade: WebSocket\r\n' +
               'Connection: Upgrade\r\n' +
               '\r\n');

  socket.pipe(socket); // echo back
});

// now that server is running
srv.listen(1337, '127.0.0.1', () => {

  // make a request
  const options = {
    port: 1337,
    hostname: '127.0.0.1',
    headers: {
      'Connection': 'Upgrade',
      'Upgrade': 'websocket'
    }
  };

  const req = http.request(options);
  req.end();

  req.on('upgrade', (res, socket, upgradeHead) => {
    console.log('got upgraded!');
    socket.end();
    process.exit(0);
  });
});
```

request.abort()

Added in: v0.3.8

Marks the request as aborting. Calling this will cause remaining data in the response to be dropped and the socket to be destroyed.

request.aborted

Added in: v0.11.14

If a request has been aborted, this value is the time when the request was aborted, in milliseconds since 1 January 1970 00:00:00 UTC.

request.connection

Added in: v0.3.0

- <net.Socket>

See request.socket

request.end([data[, encoding]][, callback])

Added in: v0.1.90

- `data` <string> | <Buffer>
- `encoding` <string>
- `callback` <Function>

Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream. If the request is chunked, this will send the terminating `'0\r\n\r\n'`.

If `data` is specified, it is equivalent to calling request.write(data, encoding) followed by `request.end(callback)`.

If `callback` is specified, it will be called when the request stream is finished.

request.flushHeaders()

Added in: v1.6.0

Flush the request headers.

For efficiency reasons, Node.js normally buffers the request headers until `request.end()` is called or the first chunk of request data is written. It then tries to pack the request headers and data into a single TCP packet.

That's usually desired (it saves a TCP round-trip), but not when the first data is not sent until possibly much later. `request.flushHeaders()` bypasses the optimization and kickstarts the request.

request.getHeader(name)

Added in: v1.6.0

- `name` <string>
- Returns: <string>

Reads out a header on the request. Note that the name is case insensitive.

Example:

```
const contentType = request.getHeader('Content-Type');
```

request.removeHeader(name)

Added in: v1.6.0

- `name` <string>

Removes a header that's already defined into headers object.

Example:

```
request.removeHeader('Content-Type');
```

request.setHeader(name, value)

Added in: v1.6.0

- `name` <string>
- `value` <string>

Sets a single header value for headers object. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here to send multiple headers with the same name.

Example:

```
request.setHeader('Content-Type', 'application/json');
```

or

```
request.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

request.setNoDelay([noDelay])

Added in: v0.5.9

- `noDelay` <boolean>

Once a socket is assigned to this request and is connected `socket.setNoDelay()` will be called.

request.setSocketKeepAlive([enable][, initialDelay])

Added in: v0.5.9

- `enable` <boolean>
- `initialDelay` <number>

Once a socket is assigned to this request and is connected `socket.setKeepAlive()` will be called.

request.setTimeout(timeout[, callback])

Added in: v0.5.9

- `timeout` <number> Milliseconds before a request times out.
- `callback` <Function> Optional function to be called when a timeout occurs. Same as binding to the `timeout` event.

Once a socket is assigned to this request and is connected `socket.setTimeout()` will be called.

Returns `request`.

request.socket

Added in: v0.3.0

- <net.Socket>

Reference to the underlying socket. Usually users will not want to access this property. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket. After `response.end()`, the property is nulled. The `socket` may also be accessed via `request.connection`.

Example:

```
const http = require('http');
const options = {
  host: 'www.google.com',
};
const req = http.get(options);
req.end();
req.once('response', (res) => {
  const ip = req.socket.localAddress;
  const port = req.socket.localPort;
  console.log(`Your IP address is ${ip} and your source port is ${port}.`);
  // consume response object
});
```

request.write(chunk[, encoding][, callback])

Added in: v0.1.29

- `chunk` <string> | <Buffer>
- `encoding` <string>
- `callback` <Function>

Sends a chunk of the body. By calling this method many times, a request body can be sent to a server — in that case it is suggested to use the `['Transfer-Encoding', 'chunked']` header line when creating the request.

The `encoding` argument is optional and only applies when `chunk` is a string. Defaults to `'utf8'`.

The `callback` argument is optional and will be called when this chunk of data is flushed.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is free again.

Class: http.Server

Added in: v0.1.17

This class inherits from net.Server and has the following additional events:

Event: 'checkContinue'

Added in: v0.3.0

- `request` <http.IncomingMessage>
- `response` <http.ServerResponse>

Emitted each time a request with an HTTP `Expect: 100-continue` is received. If this event is not listened for, the server will automatically respond with a `100 Continue` as appropriate.

Handling this event involves calling response.writeContinue() if the client should continue to send the request body, or generating an appropriate HTTP response (e.g. 400 Bad Request) if the client should not continue to send the request body.

Note that when this event is emitted and handled, the 'request' event will not be emitted.

Event: 'checkExpectation'

Added in: v5.5.0

- `request` <http.IncomingMessage>
- `response` <http.ServerResponse>

Emitted each time a request with an HTTP `Expect` header is received, where the value is not `100-continue`. If this event is not listened for, the server will automatically respond with a `417 Expectation Failed` as appropriate.

Note that when this event is emitted and handled, the 'request' event will not be emitted.

Event: 'clientError'

▸ History

- `exception` <Error>
- `socket` <net.Socket>

If a client connection emits an `'error'` event, it will be forwarded here. Listener of this event is responsible for closing/destroying the underlying socket. For example, one may wish to more gracefully close the socket with a custom HTTP response instead of abruptly severing the connection.

Default behavior is to close the socket with an HTTP '400 Bad Request' response if possible, otherwise the socket is immediately destroyed.

`socket` is the net.Socket object that the error originated from.

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end();
});
server.on('clientError', (err, socket) => {
  socket.end('HTTP/1.1 400 Bad Request\r\n\r\n');
});
server.listen(8000);
```

When the `'clientError'` event occurs, there is no `request` or `response` object, so any HTTP response sent, including response headers and payload, *must* be written directly to the `socket` object. Care must be taken to ensure the response is a properly formatted HTTP response message.

`err` is an instance of `Error` with two extra columns:

- `bytesParsed`: the bytes count of request packet that Node.js may have parsed correctly;
- `rawPacket`: the raw packet of current request.

Event: 'close'

Added in: v0.1.4

Emitted when the server closes.

Event: 'connect'

Added in: v0.7.0

- `request` <http.IncomingMessage> Arguments for the HTTP request, as it is in the 'request' event
- `socket` <net.Socket> Network socket between the server and client
- `head` <Buffer> The first packet of the tunneling stream (may be empty)

Emitted each time a client requests an HTTP `CONNECT` method. If this event is not listened for, then clients requesting a `CONNECT` method will have their connections closed.

After this event is emitted, the request's socket will not have a `'data'` event listener, meaning it will need to be bound in order to handle data sent to the server on that socket.

Event: 'connection'

Added in: v0.1.0

- `socket` <net.Socket>

This event is emitted when a new TCP stream is established. `socket` is typically an object of type net.Socket. Usually users will not want to access this event. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket. The `socket` can also be accessed at `request.connection`.

This event can also be explicitly emitted by users to inject connections into the HTTP server. In that case, any Duplex stream can be passed.

Event: 'request'

Added in: v0.1.0

- `request` <http.IncomingMessage>
- `response` <http.ServerResponse>

Emitted each time there is a request. Note that there may be multiple requests per connection (in the case of HTTP Keep-Alive connections).

Event: 'upgrade'

Added in: v0.1.94

- `request` <http.IncomingMessage> Arguments for the HTTP request, as it is in the 'request' event
- `socket` <net.Socket> Network socket between the server and client
- `head` <Buffer> The first packet of the upgraded stream (may be empty)

Emitted each time a client requests an HTTP upgrade. If this event is not listened for, then clients requesting an upgrade will have their connections closed.

After this event is emitted, the request's socket will not have a `'data'` event listener, meaning it will need to be bound in order to handle data sent to the server on that socket.

server.close([callback])

Added in: v0.1.90

- `callback` <Function>

Stops the server from accepting new connections. See net.Server.close() .

server.listen()

Starts the HTTP server listening for connections. This method is identical to server.listen() from net.Server .

server.listening

Added in: v5.7.0

- <boolean>

A Boolean indicating whether or not the server is listening for connections.

server.maxHeadersCount

Added in: v0.7.0

- <number> Defaults to 2000.

Limits maximum incoming headers count, equal to 2000 by default. If set to 0 - no limit will be applied.

server.setTimeout([msecs][, callback])

Added in: v0.9.12

- `msecs` <number> Defaults to 120000 (2 minutes).
- `callback` <Function>

Sets the timeout value for sockets, and emits a `'timeout'` event on the Server object, passing the socket as an argument, if a timeout occurs.

If there is a `'timeout'` event listener on the Server object, then it will be called with the timed-out socket as an argument.

By default, the Server's timeout value is 2 minutes, and sockets are destroyed automatically if they time out. However, if a callback is assigned to the Server's `'timeout'` event, timeouts must be handled explicitly.

Returns `server`.

server.timeout

Added in: v0.9.12

- <number> Timeout in milliseconds. Defaults to 120000 (2 minutes).

The number of milliseconds of inactivity before a socket is presumed to have timed out.

A value of `0` will disable the timeout behavior on incoming connections.

The socket timeout logic is set up on connection, so changing this value only affects new connections to the server, not any existing connections.

server.keepAliveTimeout

Added in: v8.0.0

- <number> Timeout in milliseconds. Defaults to 5000 (5 seconds).

The number of milliseconds of inactivity a server needs to wait for additional incoming data, after it has finished writing the last response, before a socket will be destroyed. If the server receives new data before the keep-alive timeout has fired, it will reset the regular inactivity timeout, i.e., `server.timeout`.

A value of `0` will disable the keep-alive timeout behavior on incoming connections. A value of `0` makes the http server behave similarly to Node.js versions prior to 8.0.0, which did not have a keep-alive timeout.

The socket timeout logic is set up on connection, so changing this value only affects new connections to the server, not any existing connections.

Class: http.ServerResponse

Added in: v0.1.17

This object is created internally by an HTTP server — not by the user. It is passed as the second parameter to the `'request'` event.

The response implements, but does not inherit from, the Writable Stream interface. This is an `EventEmitter` with the following events:

Event: 'close'

Added in: v0.6.7

Indicates that the underlying connection was terminated before `response.end()` was called or able to flush.

Event: 'finish'

Added in: v0.3.6

Emitted when the response has been sent. More specifically, this event is emitted when the last segment of the response headers and body have been handed off to the operating system for transmission over the network. It does not imply that the client has received anything yet.

After this event, no more events will be emitted on the response object.

response.addTrailers(headers)

Added in: v0.3.0

- `headers` <Object>

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Trailers will **only** be emitted if chunked encoding is used for the response; if it is not (e.g. if the request was HTTP/1.0), they will be silently discarded.

Note that HTTP requires the `Trailer` header to be sent in order to emit trailers, with a list of the header fields in its value. E.g.,

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({ 'Content-MD5': '7895bf4b8828b55ceaf47747b4bca667' });
response.end();
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

response.connection

Added in: v0.3.0

- <net.Socket>

See response.socket .

response.end([data][, encoding][, callback])

Added in: v0.1.90

- data <string> | <Buffer>
- encoding <string>
- callback <Function>

This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, response.end() , MUST be called on each response.

If data is specified, it is equivalent to calling response.write(data, encoding) followed by response.end(callback) .

If callback is specified, it will be called when the response stream is finished.

response.finished

Added in: v0.0.2

- <boolean>

Boolean value that indicates whether the response has completed. Starts as false . After response.end() executes, the value will be true .

response.getHeader(name)

Added in: v0.4.0

- name <string>
- Returns: <string>

Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive.

Example:

```
const contentType = response.getHeader('content-type');
```

response.getHeaderNames()

Added in: v7.7.0

- Returns: <Array>

Returns an array containing the unique names of the current outgoing headers. All header names are lowercase.

Example:

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headerNames = response.getHeaderNames();
// headerNames === ['foo', 'set-cookie']
```

response.getHeaders()

Added in: v7.7.0

- Returns: <Object>

Returns a shallow copy of the current outgoing headers. Since a shallow copy is used, array values may be mutated without additional calls to various header-related http module methods. The keys of the returned object are the header names and the values are the respective header values. All header names are lowercase.

The object returned by the `response.getHeaders()` method *does not* prototypically inherit from the JavaScript `Object`. This means that typical `Object` methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and *will not work*.

Example:

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headers = response.getHeaders();
// headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }
```

response.hasHeader(name)

Added in: v7.7.0

- `name` <string>
- Returns: <boolean>

Returns `true` if the header identified by `name` is currently set in the outgoing headers. Note that the header name matching is case-insensitive.

Example:

```
const hasContentType = response.hasHeader('content-type');
```

response.headersSent

Added in: v0.9.3

- <boolean>

Boolean (read-only). True if headers were sent, false otherwise.

response.removeHeader(name)

Added in: v0.4.0

- `name` <string>

Removes a header that's queued for implicit sending.

Example:

```
response.removeHeader('Content-Encoding');
```

response.sendDate

Added in: v0.7.5

- <boolean>

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

response.setHeader(name, value)

Added in: v0.4.0

- `name` <string>
- `value` <string> | <string[]>

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here to send multiple headers with the same name.

Example:

```
response.setHeader('Content-Type', 'text/html');
```

or

```
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

Attempting to set a header field name or value that contains invalid characters will result in a TypeError being thrown.

When headers have been set with response.setHeader(), they will be merged with any headers passed to response.writeHead(), with the headers passed to response.writeHead() given precedence.

```
// returns content-type = text/plain
const server = http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('ok');
});
```

response.setTimeout(msecs[, callback])

Added in: v0.9.12

- `msecs` <number>
- `callback` <Function>

Sets the Socket's timeout value to `msecs`. If a callback is provided, then it is added as a listener on the `'timeout'` event on the response object.

If no `'timeout'` listener is added to the request, the response, or the server, then sockets are destroyed when they time out. If a handler is assigned to the request, the response, or the server's `'timeout'` events, timed out sockets must be handled explicitly.

Returns `response`.

response.socket

Added in: v0.3.0

- <net.Socket>

Reference to the underlying socket. Usually users will not want to access this property. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket. After `response.end()`, the property is nulled. The `socket` may also be accessed via `response.connection`.

Example:

```
const http = require('http');
const server = http.createServer((req, res) => {
  const ip = res.socket.remoteAddress;
  const port = res.socket.remotePort;
  res.end(`Your IP address is ${ip} and your source port is ${port}.`);
}).listen(3000);
```

response.statusCode

Added in: v0.4.0

- <number>

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be sent to the client when the headers get flushed.

Example:

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code which was sent out.

response.statusMessage

Added in: v0.11.8

- <string>

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status message that will be sent to the client when the headers get flushed. If this is left as `undefined` then the standard message for the status code will be used.

Example:

```
response.statusMessage = 'Not found';
```

After response header was sent to the client, this property indicates the status message which was sent out.

response.write(chunk[, encoding][, callback])

Added in: v0.1.29

- `chunk` <string> | <Buffer>
- `encoding` <string>
- `callback` <Function>
- Returns: <boolean>

If this method is called and `response.writeHead()` has not been called, it will switch to implicit header mode and flush the implicit headers.

This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

Note that in the `http` module, the response body is omitted when the request is a HEAD request. Similarly, the `204` and `304` responses *must not* include a message body.

`chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. By default the `encoding` is `'utf8'`. `callback` will be called when this chunk of data is flushed.

This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time `response.write()` is called, it will send the buffered header information and the first chunk of the body to the client. The second time `response.write()` is called, Node.js assumes data will be streamed, and sends the new data separately. That is, the response is buffered up

to the first chunk of the body.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is free again.

response.writeContinue()

Added in: v0.3.0

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent. See the 'checkContinue' event on `Server`.

response.writeHead(statusCode[, statusMessage][, headers])

▸ History

- `statusCode` <number>
- `statusMessage` <string>
- `headers` <Object>

Sends a response header to the request. The status code is a 3-digit HTTP status code, like `404`. The last argument, `headers`, are the response headers. Optionally one can give a human-readable `statusMessage` as the second argument.

Example:

```
const body = 'hello world';
response.writeHead(200, {
  'Content-Length': Buffer.byteLength(body),
  'Content-Type': 'text/plain' });
```

This method must only be called once on a message and it must be called before response.end() is called.

If response.write() or response.end() are called before calling this, the implicit/mutable headers will be calculated and call this function.

When headers have been set with response.setHeader(), they will be merged with any headers passed to response.writeHead(), with the headers passed to response.writeHead() given precedence.

```
// returns content-type = text/plain
const server = http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('ok');
});
```

Note that Content-Length is given in bytes not characters. The above example works because the string `'hello world'` contains only single byte characters. If the body contains higher coded characters then `Buffer.byteLength()` should be used to determine the number of bytes in a given encoding. And Node.js does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

Attempting to set a header field name or value that contains invalid characters will result in a TypeError being thrown.

Class: http.IncomingMessage

Added in: v0.1.17

An `IncomingMessage` object is created by http.Server or http.ClientRequest and passed as the first argument to the 'request' and 'response' event respectively. It may be used to access response status, headers and data.

It implements the Readable Stream interface, as well as the following additional events, methods, and properties.

Event: 'aborted'

Added in: v0.3.8

Emitted when the request has been aborted and the network socket has closed.

Event: 'close'

Added in: v0.4.2

Indicates that the underlying connection was closed. Just like `'end'`, this event occurs only once per response.

message.destroy([error])

Added in: v0.3.0

- `error` <Error>

Calls `destroy()` on the socket that received the `IncomingMessage`. If `error` is provided, an `'error'` event is emitted and `error` is passed as an argument to any listeners on the event.

message.headers

Added in: v0.1.5

- <Object>

The request/response headers object.

Key-value pairs of header names and values. Header names are lower-cased. Example:

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

Duplicates in raw headers are handled in the following ways, depending on the header name:

- Duplicates of `age`, `authorization`, `content-length`, `content-type`, `etag`, `expires`, `from`, `host`, `if-modified-since`, `if-unmodified-since`, `last-modified`, `location`, `max-forwards`, `proxy-authorization`, `referer`, `retry-after`, or `user-agent` are discarded.
- `set-cookie` is always an array. Duplicates are added to the array.
- For all other headers, the values are joined together with ', '.

message.httpVersion

Added in: v0.1.1

- <string>

In case of server request, the HTTP version sent by the client. In the case of client response, the HTTP version of the connected-to server. Probably either `'1.1'` or `'1.0'`.

Also `message.httpVersionMajor` is the first integer and `message.httpVersionMinor` is the second.

message.method

Added in: v0.1.1

- <string>

Only valid for request obtained from `http.Server`.

The request method as a string. Read only. Example: `'GET'`, `'DELETE'`.

message.rawHeaders

Added in: v0.11.6

- <Array>

The raw request/response headers list exactly as they were received.

Note that the keys and values are in the same list. It is *not* a list of tuples. So, the even-numbered offsets are key values, and the odd-numbered offsets are the associated values.

Header names are not lowercased, and duplicates are not merged.

```
// Prints something like:
//
// [ 'user-agent',
//   'this is invalid because there can be only one',
//   'User-Agent',
//   'curl/7.22.0',
//   'Host',
//   '127.0.0.1:8000',
//   'ACCEPT',
//   '*/*' ]
console.log(request.rawHeaders);
```

message.rawTrailers

Added in: v0.11.6

- <Array>

The raw request/response trailer keys and values exactly as they were received. Only populated at the `'end'` event.

message.setTimeout(msecs, callback)

Added in: v0.5.9

- `msecs` <number>
- `callback` <Function>

Calls `message.connection.setTimeout(msecs, callback)`.

Returns `message`.

message.socket

Added in: v0.3.0

- <net.Socket>

The net.Socket object associated with the connection.

With HTTPS support, use request.socket.getPeerCertificate() to obtain the client's authentication details.

message.statusCode

Added in: v0.1.1

- <number>

Only valid for response obtained from http.ClientRequest.

The 3-digit HTTP response status code. E.G. `404`.

message.statusMessage

Added in: v0.11.10

- <string>

Only valid for response obtained from `http.ClientRequest`.

The HTTP response status message (reason phrase). E.G. `OK` or `Internal Server Error`.

message.trailers

Added in: v0.3.0

- <Object>

The request/response trailers object. Only populated at the `'end'` event.

message.url

Added in: v0.1.90

- <string>

Only valid for request obtained from `http.Server`.

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

Then `request.url` will be:

```
'/status?name=ryan'
```

To parse the url into its parts `require('url').parse(request.url)` can be used. Example:

```
$ node
> require('url').parse('/status?name=ryan')
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status',
  path: '/status?name=ryan',
  href: '/status?name=ryan' }
```

To extract the parameters from the query string, the `require('querystring').parse` function can be used, or `true` can be passed as the second argument to `require('url').parse`. Example:

```
$ node
> require('url').parse('/status?name=ryan', true)
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=ryan',
  query: { name: 'ryan' },
  pathname: '/status',
  path: '/status?name=ryan',
  href: '/status?name=ryan' }
```

http.METHODS

Added in: v0.11.8

- <Array>

A list of the HTTP methods that are supported by the parser.

http.STATUS_CODES

Added in: v0.1.22

- <Object>

A collection of all the standard HTTP response status codes, and the short description of each. For example, `http.STATUS_CODES[404] === 'Not Found'`.

http.createServer([options][, requestListener])

▸ History

- `options` <Object>
  - `IncomingMessage` <http.IncomingMessage> Specifies the IncomingMessage class to be used. Useful for extending the original `IncomingMessage`. Defaults to: `IncomingMessage`
  - `ServerResponse` <http.ServerResponse> Specifies the ServerResponse class to be used. Useful for extending the original `ServerResponse`. Defaults to: `ServerResponse`
- `requestListener` <Function>
- Returns: <http.Server>

Returns a new instance of `http.Server`.

The `requestListener` is a function which is automatically added to the `'request'` event.

http.get(options[, callback])

▸ History

- `options` <Object> | <string> | <URL> Accepts the same `options` as `http.request()`, with the `method` always set to `GET`. Properties that are inherited from the prototype are ignored.
- `callback` <Function>
- Returns: <http.ClientRequest>

Since most requests are GET requests without bodies, Node.js provides this convenience method. The only difference between this method and `http.request()` is that it sets the method to GET and calls `req.end()` automatically. Note that the callback must take care to consume the response data for reasons stated in `http.ClientRequest` section.

The `callback` is invoked with a single argument that is an instance of `http.IncomingMessage`

JSON Fetching Example:

```
http.get('http://nodejs.org/dist/index.json', (res) => {
  const { statusCode } = res;
  const contentType = res.headers['content-type'];

  let error;
  if (statusCode !== 200) {
    error = new Error('Request Failed.\n' +
                      `Status Code: ${statusCode}`);
  } else if (!/^application\/json/.test(contentType)) {
    error = new Error('Invalid content-type.\n' +
                      `Expected application/json but received ${contentType}`);
  }
  if (error) {
    console.error(error.message);
    // consume response data to free up memory
    res.resume();
    return;
  }

  res.setEncoding('utf8');
  let rawData = '';
  res.on('data', (chunk) => { rawData += chunk; });
  res.on('end', () => {
    try {
      const parsedData = JSON.parse(rawData);
      console.log(parsedData);
    } catch (e) {
      console.error(e.message);
    }
  });
}).on('error', (e) => {
  console.error(`Got error: ${e.message}`);
});
```

http.globalAgent

Added in: v0.5.9

- <http.Agent>

Global instance of `Agent` which is used as the default for all HTTP client requests.

http.request(options[, callback])

▸ History

- `options` <Object> | <string> | <URL>
  - `protocol` <string> Protocol to use. Defaults to `http:`.
  - `host` <string> A domain name or IP address of the server to issue the request to. Defaults to `localhost`.
  - `hostname` <string> Alias for `host`. To support `url.parse()`, `hostname` is preferred over `host`.

- `family` <number> IP address family to use when resolving `host` and `hostname`. Valid values are `4` or `6`. When unspecified, both IP v4 and v6 will be used.
- `port` <number> Port of remote server. Defaults to 80.
- `localAddress` <string> Local interface to bind for network connections.
- `socketPath` <string> Unix Domain Socket (use one of host:port or socketPath).
- `method` <string> A string specifying the HTTP request method. Defaults to `'GET'`.
- `path` <string> Request path. Defaults to `'/'`. Should include query string if any. E.G. `'/index.html?page=12'`. An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may change in the future.
- `headers` <Object> An object containing request headers.
- `auth` <string> Basic authentication i.e. `'user:password'` to compute an Authorization header.
- `agent` <http.Agent> | <boolean> Controls `Agent` behavior. Possible values:
  - `undefined` (default): use `http.globalAgent` for this host and port.
  - `Agent` object: explicitly use the passed in `Agent`.
  - `false`: causes a new `Agent` with default values to be used.
- `createConnection` <Function> A function that produces a socket/stream to use for the request when the `agent` option is not used. This can be used to avoid creating a custom `Agent` class just to override the default `createConnection` function. See `agent.createConnection()` for more details. Any `Duplex` stream is a valid return value.
- `timeout` <number>: A number specifying the socket timeout in milliseconds. This will set the timeout before the socket is connected.
- `setHost` <boolean>: Specifies whether or not to automatically add the `Host` header. Defaults to `true`.

* `callback` <Function>
* Returns: <http.ClientRequest>

Node.js maintains several connections per server to make HTTP requests. This function allows one to transparently issue requests.

`options` can be an object, a string, or a `URL` object. If `options` is a string, it is automatically parsed with `url.parse()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

The optional `callback` parameter will be added as a one-time listener for the `'response'` event.

`http.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.

Example:

```
const postData = querystring.stringify({
  'msg': 'Hello World!'
});

const options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': Buffer.byteLength(postData)
  }
};

const req = http.request(options, (res) => {
  console.log(`STATUS: ${res.statusCode}`);
  console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(`BODY: ${chunk}`);
  });
  res.on('end', () => {
    console.log('No more data in response.');
  });
});

req.on('error', (e) => {
  console.error(`problem with request: ${e.message}`);
});

// write data to request body
req.write(postData);
req.end();
```

Note that in the example `req.end()` was called. With `http.request()` one must always call `req.end()` to signify the end of the request - even if there is no data being written to the request body.

If any error is encountered during the request (be that with DNS resolution, TCP level errors, or actual HTTP parse errors) an `'error'` event is emitted on the returned request object. As with all `'error'` events, if no listeners are registered the error will be thrown.

There are a few special headers that should be noted.

- Sending a 'Connection: keep-alive' will notify Node.js that the connection to the server should be persisted until the next request.
- Sending a 'Content-Length' header will disable the default chunked encoding.
- Sending an 'Expect' header will immediately send the request headers. Usually, when sending 'Expect: 100-continue', both a timeout and a listener for the `continue` event should be set. See RFC2616 Section 8.2.3 for more information.
- Sending an Authorization header will override using the `auth` option to compute basic authentication.

Example using a `URL` as `options`:

```
const { URL } = require('url');

const options = new URL('http://abc:xyz@example.com');

const req = http.request(options, (res) => {
  // ...
});
```

In a successful request, the following events will be emitted in the following order:

- `socket`
- `response`
    - `data` any number of times, on the `res` object (`data` will not be emitted at all if the response body is empty, for instance, in most redirects)
    - `end` on the `res` object
- `close`

In the case of a connection error, the following events will be emitted:

- `socket`
- `error`
- `close`

If `req.abort()` is called before the connection succeeds, the following events will be emitted in the following order:

- `socket`
- (`req.abort()` called here)
- `abort`
- `close`
- `error` with an error with message `Error: socket hang up` and code `ECONNRESET`

If `req.abort()` is called after the response is received, the following events will be emitted in the following order:

- `socket`
- `response`
    - `data` any number of times, on the `res` object
- (`req.abort()` called here)
- `abort`
- `close`
    - `aborted` on the `res` object
    - `end` on the `res` object
    - `close` on the `res` object

Note that setting the `timeout` option or using the `setTimeout` function will not abort the request or do anything besides add a `timeout` event.

HTTP/2

Stability: 1 - Experimental

The `http2` module provides an implementation of the HTTP/2 protocol. It can be accessed using:

```
const http2 = require('http2');
```

Core API

The Core API provides a low-level interface designed specifically around support for HTTP/2 protocol features. It is specifically *not* designed for compatibility with the existing HTTP/1 module API. However, the Compatibility API is.

The `http2` Core API is much more symmetric between client and server than the `http` API. For instance, most events, like `error`, `connect` and `stream`, can be emitted either by client-side code or server-side code.

Server-side example

The following illustrates a simple HTTP/2 server using the Core API. Since there are no browsers known that support unencrypted HTTP/2, the use of `http2.createSecureServer()` is necessary when communicating with browser clients.

```
const http2 = require('http2');
const fs = require('fs');

const server = http2.createSecureServer({
  key: fs.readFileSync('localhost-privkey.pem'),
  cert: fs.readFileSync('localhost-cert.pem')
});
server.on('error', (err) => console.error(err));

server.on('stream', (stream, headers) => {
  // stream is a Duplex
  stream.respond({
    'content-type': 'text/html',
    ':status': 200
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8443);
```

To generate the certificate and key for this example, run:

```
openssl req -x509 -newkey rsa:2048 -nodes -sha256 -subj '/CN=localhost' \
  -keyout localhost-privkey.pem -out localhost-cert.pem
```

Client-side example

The following illustrates an HTTP/2 client:

```
const http2 = require('http2');
const fs = require('fs');
const client = http2.connect('https://localhost:8443', {
  ca: fs.readFileSync('localhost-cert.pem')
});
client.on('error', (err) => console.error(err));

const req = client.request({ ':path': '/' });

req.on('response', (headers, flags) => {
  for (const name in headers) {
    console.log(`${name}: ${headers[name]}`);
  }
});

req.setEncoding('utf8');
let data = '';
req.on('data', (chunk) => { data += chunk; });
req.on('end', () => {
  console.log(`\n${data}`);
  client.close();
});
req.end();
```

Class: Http2Session

Added in: v8.4.0

- Extends: <EventEmitter>

Instances of the `http2.Http2Session` class represent an active communications session between an HTTP/2 client and server. Instances of this class are *not* intended to be constructed directly by user code.

Each `Http2Session` instance will exhibit slightly different behaviors depending on whether it is operating as a server or a client. The `http2session.type` property can be used to determine the mode in which an `Http2Session` is operating. On the server side, user code should rarely have occasion to work with the `Http2Session` object directly, with most actions typically taken through interactions with either the `Http2Server` or `Http2Stream` objects.

Http2Session and Sockets

Every `Http2Session` instance is associated with exactly one net.Socket or tls.TLSSocket when it is created. When either the `Socket` or the `Http2Session` are destroyed, both will be destroyed.

Because the of the specific serialization and processing requirements imposed by the HTTP/2 protocol, it is not recommended for user code to read data from or write data to a `Socket` instance bound to a `Http2Session`. Doing so can put the HTTP/2 session into an indeterminate state causing the session and the socket to become unusable.

Once a `Socket` has been bound to an `Http2Session`, user code should rely solely on the API of the `Http2Session`.

Event: 'close'

Added in: v8.4.0

The `'close'` event is emitted once the `Http2Session` has been destroyed.

Event: 'connect'

Added in: v8.4.0

The `'connect'` event is emitted once the `Http2Session` has been successfully connected to the remote peer and communication may begin.

User code will typically not listen for this event directly.

Event: 'error'

Added in: v8.4.0

The `'error'` event is emitted when an error occurs during the processing of an `Http2Session`.

Event: 'frameError'

Added in: v8.4.0

The `'frameError'` event is emitted when an error occurs while attempting to send a frame on the session. If the frame that could not be sent is associated with a specific `Http2Stream`, an attempt to emit `'frameError'` event on the `Http2Stream` is made.

When invoked, the handler function will receive three arguments:

- An integer identifying the frame type.
- An integer identifying the error code.
- An integer identifying the stream (or 0 if the frame is not associated with a stream).

If the `'frameError'` event is associated with a stream, the stream will be closed and destroyed immediately following the `'frameError'` event. If the event is not associated with a stream, the `Http2Session` will be shut down immediately following the `'frameError'` event.

Event: 'goaway'

Added in: v8.4.0

The `'goaway'` event is emitted when a GOAWAY frame is received. When invoked, the handler function will receive three arguments:

- `errorCode` <number> The HTTP/2 error code specified in the GOAWAY frame.
- `lastStreamID` <number> The ID of the last stream the remote peer successfully processed (or `0` if no ID is specified).
- `opaqueData` <Buffer> If additional opaque data was included in the GOAWAY frame, a `Buffer` instance will be passed containing that data.

The `Http2Session` instance will be shut down automatically when the `'goaway'` event is emitted.

Event: 'localSettings'

Added in: v8.4.0

The `'localSettings'` event is emitted when an acknowledgment SETTINGS frame has been received. When invoked, the handler function will receive a copy of the local settings.

When using `http2session.settings()` to submit new settings, the modified settings do not take effect until the `'localSettings'` event is emitted.

```
session.settings({ enablePush: false });

session.on('localSettings', (settings) => {
  /** use the new settings **/
});
```

Event: 'remoteSettings'

Added in: v8.4.0

The `'remoteSettings'` event is emitted when a new SETTINGS frame is received from the connected peer. When invoked, the handler function will receive a copy of the remote settings.

```
session.on('remoteSettings', (settings) => {
  /** use the new settings **/
});
```

Event: 'stream'

Added in: v8.4.0

The 'stream' event is emitted when a new Http2Stream is created. When invoked, the handler function will receive a reference to the Http2Stream object, a HTTP/2 Headers Object, and numeric flags associated with the creation of the stream.

```
const http2 = require('http2');
session.on('stream', (stream, headers, flags) => {
  const method = headers[':method'];
  const path = headers[':path'];
  // ...
  stream.respond({
    ':status': 200,
    'content-type': 'text/plain'
  });
  stream.write('hello ');
  stream.end('world');
});
```

On the server side, user code will typically not listen for this event directly, and would instead register a handler for the 'stream' event emitted by the net.Server or tls.Server instances returned by http2.createServer() and http2.createSecureServer(), respectively, as in the example below:

```
const http2 = require('http2');

// Create an unencrypted HTTP/2 server
const server = http2.createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html',
    ':status': 200
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(80);
```

Event: 'timeout'

Added in: v8.4.0

After the http2session.setTimeout() method is used to set the timeout period for this Http2Session, the 'timeout' event is emitted if there is no activity on the Http2Session after the configured number of milliseconds.

```
session.setTimeout(2000);
session.on('timeout', () => { /** .. **/ });
```

http2session.alpnProtocol

Added in: v9.4.0

- Value: <string> | <undefined>

Value will be undefined if the Http2Session is not yet connected to a socket, h2c if the Http2Session is not connected to a TLSSocket, or will return the value of the connected TLSSocket's own alpnProtocol property.

http2session.close([callback])

Added in: v9.4.0

- `callback` <Function>

Gracefully closes the `Http2Session`, allowing any existing streams to complete on their own and preventing new `Http2Stream` instances from being created. Once closed, `http2session.destroy()` *might* be called if there are no open `Http2Stream` instances.

If specified, the `callback` function is registered as a handler for the `'close'` event.

http2session.closed

Added in: v9.4.0

- Value: <boolean>

Will be `true` if this `Http2Session` instance has been closed, otherwise `false`.

http2session.destroy([error,][code])

Added in: v8.4.0

- `error` <Error> An `Error` object if the `Http2Session` is being destroyed due to an error.
- `code` <number> The HTTP/2 error code to send in the final `GOAWAY` frame. If unspecified, and `error` is not undefined, the default is `INTERNAL_ERROR`, otherwise defaults to `NO_ERROR`.

Immediately terminates the `Http2Session` and the associated `net.Socket` or `tls.TLSSocket`.

Once destroyed, the `Http2Session` will emit the `'close'` event. If `error` is not undefined, an `'error'` event will be emitted immediately after the `'close'` event.

If there are any remaining open `Http2Streams` associated with the `Http2Session`, those will also be destroyed.

http2session.destroyed

Added in: v8.4.0

- Value: <boolean>

Will be `true` if this `Http2Session` instance has been destroyed and must no longer be used, otherwise `false`.

http2session.encrypted

Added in: v9.4.0

- Value: <boolean> | <undefined>

Value is `undefined` if the `Http2Session` session socket has not yet been connected, `true` if the `Http2Session` is connected with a `TLSSocket`, and `false` if the `Http2Session` is connected to any other kind of socket or stream.

http2session.goaway([code, [lastStreamID, [opaqueData]]])

Added in: v9.4.0

- `code` <number> An HTTP/2 error code
- `lastStreamID` <number> The numeric ID of the last processed `Http2Stream`
- `opaqueData` <Buffer> | <TypedArray> | <DataView> A `TypedArray` or `DataView` instance containing additional data to be carried within the GOAWAY frame.

Transmits a `GOAWAY` frame to the connected peer *without* shutting down the `Http2Session`.

http2session.localSettings

Added in: v8.4.0

- Value: <HTTP/2 Settings Object>

A prototype-less object describing the current local settings of this `Http2Session`. The local settings are local to *this* `Http2Session` instance.

http2session.originSet

Added in: v9.4.0

- Value: <string[]> | <undefined>

If the `Http2Session` is connected to a `TLSSocket`, the `originSet` property will return an Array of origins for which the `Http2Session` may be considered authoritative.

http2session.pendingSettingsAck

Added in: v8.4.0

- Value: <boolean>

Indicates whether or not the `Http2Session` is currently waiting for an acknowledgment for a sent SETTINGS frame. Will be `true` after calling the `http2session.settings()` method. Will be `false` once all sent SETTINGS frames have been acknowledged.

http2session.ping([payload,]callback)

Added in: v9.2.1

- `payload` <Buffer> | <TypedArray> | <DataView> Optional ping payload.
- `callback` <Function>
- Returns: <boolean>

Sends a `PING` frame to the connected HTTP/2 peer. A `callback` function must be provided. The method will return `true` if the `PING` was sent, `false` otherwise.

The maximum number of outstanding (unacknowledged) pings is determined by the `maxOutstandingPings` configuration option. The default maximum is 10.

If provided, the `payload` must be a `Buffer`, `TypedArray`, or `DataView` containing 8 bytes of data that will be transmitted with the `PING` and returned with the ping acknowledgment.

The callback will be invoked with three arguments: an error argument that will be `null` if the `PING` was successfully acknowledged, a `duration` argument that reports the number of milliseconds elapsed since the ping was sent and the acknowledgment was received, and a `Buffer` containing the 8-byte `PING` payload.

```
session.ping(Buffer.from('abcdefgh'), (err, duration, payload) => {
  if (!err) {
    console.log(`Ping acknowledged in ${duration} milliseconds`);
    console.log(`With payload '${payload.toString()}'`);
  }
});
```

If the `payload` argument is not specified, the default payload will be the 64-bit timestamp (little endian) marking the start of the `PING` duration.

http2session.ref()

Added in: v9.4.0

Calls ref() on this `Http2Session` instance's underlying net.Socket.

http2session.remoteSettings

Added in: v8.4.0

- Value: <HTTP/2 Settings Object>

A prototype-less object describing the current remote settings of this `Http2Session`. The remote settings are set by the *connected* HTTP/2 peer.

http2session.setTimeout(msecs, callback)

Added in: v8.4.0

- `msecs` <number>
- `callback` <Function>

Used to set a callback function that is called when there is no activity on the `Http2Session` after `msecs` milliseconds. The given `callback` is registered as a listener on the `'timeout'` event.

http2session.socket

Added in: v8.4.0

- Value: <net.Socket> | <tls.TLSSocket>

Returns a Proxy object that acts as a `net.Socket` (or `tls.TLSSocket`) but limits available methods to ones safe to use with HTTP/2.

`destroy`, `emit`, `end`, `pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See Http2Session and Sockets for more information.

`setTimeout` method will be called on this `Http2Session`.

All other interactions will be routed directly to the socket.

http2session.state

Added in: v8.4.0

Provides miscellaneous information about the current state of the `Http2Session`.

- Value: <Object>
  - `effectiveLocalWindowSize` <number> The current local (receive) flow control window size for the `Http2Session`.
  - `effectiveRecvDataLength` <number> The current number of bytes that have been received since the last flow control `WINDOW_UPDATE`.
  - `nextStreamID` <number> The numeric identifier to be used the next time a new `Http2Stream` is created by this `Http2Session`.
  - `localWindowSize` <number> The number of bytes that the remote peer can send without receiving a `WINDOW_UPDATE`.
  - `lastProcStreamID` <number> The numeric id of the `Http2Stream` for which a `HEADERS` or `DATA` frame was most recently received.
  - `remoteWindowSize` <number> The number of bytes that this `Http2Session` may send without receiving a `WINDOW_UPDATE`.
  - `outboundQueueSize` <number> The number of frames currently within the outbound queue for this `Http2Session`.
  - `deflateDynamicTableSize` <number> The current size in bytes of the outbound header compression state table.
  - `inflateDynamicTableSize` <number> The current size in bytes of the inbound header compression state table.

An object describing the current status of this `Http2Session`.

http2session.settings(settings)

Added in: v8.4.0

- `settings` <HTTP/2 Settings Object>

Updates the current local settings for this `Http2Session` and sends a new `SETTINGS` frame to the connected HTTP/2 peer.

Once called, the `http2session.pendingSettingsAck` property will be `true` while the session is waiting for the remote peer to acknowledge the new settings.

The new settings will not become effective until the SETTINGS acknowledgment is received and the `'localSettings'` event is emitted. It is possible to send multiple SETTINGS frames while acknowledgment is still pending.

http2session.type

Added in: v8.4.0

- Value: <number>

The http2session.type will be equal to http2.constants.NGHTTP2_SESSION_SERVER if this Http2Session instance is a server, and http2.constants.NGHTTP2_SESSION_CLIENT if the instance is a client.

http2session.unref()

Added in: v9.4.0

Calls unref() on this Http2Session instance's underlying net.Socket .

Class: ServerHttp2Session

Added in: v8.4.0

serverhttp2session.altsvc(alt, originOrStream)

Added in: v9.4.0

- alt <string> A description of the alternative service configuration as defined by RFC 7838 .
- originOrStream <number> | <string> | <URL> | <Object> Either a URL string specifying the origin (or an Object with an origin property) or the numeric identifier of an active Http2Stream as given by the http2stream.id property.

Submits an ALTSVC frame (as defined by RFC 7838) to the connected client.

```
const http2 = require('http2');

const server = http2.createServer();
server.on('session', (session) => {
  // Set altsvc for origin https://example.org:80
  session.altsvc('h2=":8000"', 'https://example.org:80');
});

server.on('stream', (stream) => {
  // Set altsvc for a specific stream
  stream.session.altsvc('h2=":8000"', stream.id);
});
```

Sending an ALTSVC frame with a specific stream ID indicates that the alternate service is associated with the origin of the given Http2Stream .

The alt and origin string *must* contain only ASCII bytes and are strictly interpreted as a sequence of ASCII bytes. The special value 'clear' may be passed to clear any previously set alternative service for a given domain.

When a string is passed for the originOrStream argument, it will be parsed as a URL and the origin will be derived. For instance, the origin for the HTTP URL 'https://example.org/foo/bar' is the ASCII string 'https://example.org' . An error will be thrown if either the given string cannot be parsed as a URL or if a valid origin cannot be derived.

A URL object, or any object with an origin property, may be passed as originOrStream , in which case the value of the origin property will be used. The value of the origin property *must* be a properly serialized ASCII origin.

Specifying alternative services

The format of the alt parameter is strictly defined by RFC 7838 as an ASCII string containing a comma-delimited list of "alternative" protocols associated with a specific host and port.

For example, the value 'h2="example.org:81"' indicates that the HTTP/2 protocol is available on the host 'example.org' on TCP/IP port 81. The host and port *must* be contained within the quote (") characters.

Multiple alternatives may be specified, for instance: 'h2="example.org:81", h2=":82"'

The protocol identifier ('h2' in the examples) may be any valid ALPN Protocol ID .

The syntax of these values is not validated by the Node.js implementation and are passed through as provided by the user or received from the peer.

Class: ClientHttp2Session

Added in: v8.4.0

Event: 'altsvc'

Added in: v9.4.0

- `alt` : <string>
- `origin` : <string>
- `streamId` : <number>

The `'altsvc'` event is emitted whenever an `ALTSVC` frame is received by the client. The event is emitted with the `ALTSVC` value, origin, and stream ID. If no `origin` is provided in the `ALTSVC` frame, `origin` will be an empty string.

```
const http2 = require('http2');
const client = http2.connect('https://example.org');

client.on('altsvc', (alt, origin, streamId) => {
  console.log(alt);
  console.log(origin);
  console.log(streamId);
});
```

clienthttp2session.request(headers[, options])

Added in: v8.4.0

- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `endStream` <boolean> `true` if the `Http2Stream` *writable* side should be closed initially, such as when sending a `GET` request that should not expect a payload body.
  - `exclusive` <boolean> When `true` and `parent` identifies a parent Stream, the created stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of the newly created stream. **Default:** `false`
  - `parent` <number> Specifies the numeric identifier of a stream the newly created stream is dependent on.
  - `weight` <number> Specifies the relative dependency of a stream in relation to other streams with the same `parent`. The value is a number between `1` and `256` (inclusive).
  - `getTrailers` <Function> Callback function invoked to collect trailer headers.
- Returns: <ClientHttp2Stream>

For HTTP/2 Client `Http2Session` instances only, the `http2session.request()` creates and returns an `Http2Stream` instance that can be used to send an HTTP/2 request to the connected server.

This method is only available if `http2session.type` is equal to `http2.constants.NGHTTP2_SESSION_CLIENT`.

```js
const http2 = require('http2');
const clientSession = http2.connect('https://localhost:1234');
const {
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS
} = http2.constants;

const req = clientSession.request({ [HTTP2_HEADER_PATH]: '/' });
req.on('response', (headers) => {
  console.log(headers[HTTP2_HEADER_STATUS]);
  req.on('data', (chunk) => { /** .. **/ });
  req.on('end', () => { /** .. **/ });
});
```

When set, the `options.getTrailers()` function is called immediately after queuing the last chunk of payload data to be sent. The callback is passed a single object (with a `null` prototype) that the listener may use to specify the trailing header fields to send to the peer.

The HTTP/1 specification forbids trailers from containing HTTP/2 pseudo-header fields (e.g. `':method'`, `':path'`, etc). An `'error'` event will be emitted if the `getTrailers` callback attempts to set such header fields.

The `:method` and `:path` pseudo-headers are not specified within `headers`, they respectively default to:

- `:method` = `'GET'`
- `:path` = `/`

Class: Http2Stream

Added in: v8.4.0

- Extends: <stream.Duplex>

Each instance of the `Http2Stream` class represents a bidirectional HTTP/2 communications stream over an `Http2Session` instance. Any single `Http2Session` may have up to 2^{31}-1 `Http2Stream` instances over its lifetime.

User code will not construct `Http2Stream` instances directly. Rather, these are created, managed, and provided to user code through the `Http2Session` instance. On the server, `Http2Stream` instances are created either in response to an incoming HTTP request (and handed off to user code via the `'stream'` event), or in response to a call to the `http2stream.pushStream()` method. On the client, `Http2Stream` instances are created and returned when either the `http2session.request()` method is called, or in response to an incoming `'push'` event.

The `Http2Stream` class is a base for the ServerHttp2Stream and ClientHttp2Stream classes, each of which is used specifically by either the Server or Client side, respectively.

All `Http2Stream` instances are Duplex streams. The `Writable` side of the `Duplex` is used to send data to the connected peer, while the `Readable` side is used to receive data sent by the connected peer.

Http2Stream Lifecycle

Creation

On the server side, instances of ServerHttp2Stream are created either when:

- A new HTTP/2 `HEADERS` frame with a previously unused stream ID is received;
- The `http2stream.pushStream()` method is called.

On the client side, instances of ClientHttp2Stream are created when the `http2session.request()` method is called.

On the client, the `Http2Stream` instance returned by `http2session.request()` may not be immediately ready for use if the parent `Http2Session` has not yet been fully established. In such cases, operations called on the `Http2Stream` will be buffered until the `'ready'` event is emitted. User code should rarely, if ever, need to handle the `'ready'` event directly. The ready status of an `Http2Stream` can be determined by checking the value of `http2stream.id`. If the value is `undefined`, the stream is not yet ready for use.

Destruction

All Http2Stream instances are destroyed either when:

- An `RST_STREAM` frame for the stream is received by the connected peer.
- The `http2stream.close()` method is called.
- The `http2stream.destroy()` or `http2session.destroy()` methods are called.

When an `Http2Stream` instance is destroyed, an attempt will be made to send an `RST_STREAM` frame will be sent to the connected peer.

When the `Http2Stream` instance is destroyed, the `'close'` event will be emitted. Because `Http2Stream` is an instance of `stream.Duplex`, the `'end'` event will also be emitted if the stream data is currently flowing. The `'error'` event may also be emitted if `http2stream.destroy()` was called with an `Error` passed as the first argument.

After the `Http2Stream` has been destroyed, the `http2stream.destroyed` property will be `true` and the `http2stream.rstCode` property will specify the `RST_STREAM` error code. The `Http2Stream` instance is no longer usable once destroyed.

Event: 'aborted'

Added in: v8.4.0

The `'aborted'` event is emitted whenever a `Http2Stream` instance is abnormally aborted in mid-communication.

The `'aborted'` event will only be emitted if the `Http2Stream` writable side has not been ended.

Event: 'close'

Added in: v8.4.0

The `'close'` event is emitted when the `Http2Stream` is destroyed. Once this event is emitted, the `Http2Stream` instance is no longer usable.

The listener callback is passed a single argument specifying the HTTP/2 error code specified when closing the stream. If the code is any value other than `NGHTTP2_NO_ERROR` (`0`), an `'error'` event will also be emitted.

Event: 'error'

Added in: v8.4.0

The `'error'` event is emitted when an error occurs during the processing of an `Http2Stream`.

Event: 'frameError'

Added in: v8.4.0

The `'frameError'` event is emitted when an error occurs while attempting to send a frame. When invoked, the handler function will receive an integer argument identifying the frame type, and an integer argument identifying the error code. The `Http2Stream` instance will be destroyed immediately after the `'frameError'` event is emitted.

Event: 'timeout'

Added in: v8.4.0

The `'timeout'` event is emitted after no activity is received for this `'Http2Stream'` within the number of milliseconds set using `http2stream.setTimeout()`.

Event: 'trailers'

Added in: v8.4.0

The `'trailers'` event is emitted when a block of headers associated with trailing header fields is received. The listener callback is passed the HTTP/2 Headers Object and flags associated with the headers.

```
stream.on('trailers', (headers, flags) => {
  console.log(headers);
});
```

http2stream.aborted

Added in: v8.4.0

- Value: <boolean>

Set to `true` if the `Http2Stream` instance was aborted abnormally. When set, the `'aborted'` event will have been emitted.

http2stream.close(code[, callback])

Added in: v8.4.0

- code <number> Unsigned 32-bit integer identifying the error code. **Default:** `http2.constants.NGHTTP2_NO_ERROR` (`0x00`)
- `callback` <Function> An optional function registered to listen for the `'close'` event.

Closes the `Http2Stream` instance by sending an `RST_STREAM` frame to the connected HTTP/2 peer.

http2stream.closed

Added in: v9.4.0

- Value: <boolean>

Set to `true` if the `Http2Stream` instance has been closed.

http2stream.destroyed

Added in: v8.4.0

- Value: <boolean>

Set to `true` if the `Http2Stream` instance has been destroyed and is no longer usable.

http2stream.pending

Added in: v9.4.0

- Value: <boolean>

Set to `true` if the `Http2Stream` instance has not yet been assigned a numeric stream identifier.

http2stream.priority(options)

Added in: v8.4.0

- `options` <Object>
  - `exclusive` <boolean> When `true` and `parent` identifies a parent Stream, this stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of this stream. **Default:** `false`
  - `parent` <number> Specifies the numeric identifier of a stream this stream is dependent on.
  - `weight` <number> Specifies the relative dependency of a stream in relation to other streams with the same `parent`. The value is a number between `1` and `256` (inclusive).
  - `silent` <boolean> When `true`, changes the priority locally without sending a `PRIORITY` frame to the connected peer.

Updates the priority for this `Http2Stream` instance.

http2stream.rstCode

Added in: v8.4.0

- Value: <number>

Set to the `RST_STREAM` error code reported when the `Http2Stream` is destroyed after either receiving an `RST_STREAM` frame from the

connected peer, calling `http2stream.close()`, or `http2stream.destroy()`. Will be `undefined` if the `Http2Stream` has not been closed.

http2stream.sentHeaders

Added in: v9.5.0

- Value: <HTTP/2 Headers Object>

An object containing the outbound headers sent for this `Http2Stream`.

http2stream.sentInfoHeaders

Added in: v9.5.0

- Value: <HTTP/2 Headers Object[]>

An array of objects containing the outbound informational (additional) headers sent for this `Http2Stream`.

http2stream.sentTrailers

Added in: v9.5.0

- Value: <HTTP/2 Headers Object>

An object containing the outbound trailers sent for this this `HttpStream`.

http2stream.session

Added in: v8.4.0

- Value: <Http2Session>

A reference to the `Http2Session` instance that owns this `Http2Stream`. The value will be `undefined` after the `Http2Stream` instance is destroyed.

http2stream.setTimeout(msecs, callback)

Added in: v8.4.0

- `msecs` <number>
- `callback` <Function>

```
const http2 = require('http2');
const client = http2.connect('http://example.org:8000');
const { NGHTTP2_CANCEL } = http2.constants;
const req = client.request({ ':path': '/' });

// Cancel the stream if there's no activity after 5 seconds
req.setTimeout(5000, () => req.close(NGHTTP2_CANCEL));
```

http2stream.state

Added in: v8.4.0

Provides miscellaneous information about the current state of the `Http2Stream`.

- Value: <Object>
  - `localWindowSize` <number> The number of bytes the connected peer may send for this `Http2Stream` without receiving a `WINDOW_UPDATE`.
  - `state` <number> A flag indicating the low-level current state of the `Http2Stream` as determined by nghttp2.
  - `localClose` <number> `true` if this `Http2Stream` has been closed locally.
  - `remoteClose` <number> `true` if this `Http2Stream` has been closed remotely.
  - `sumDependencyWeight` <number> The sum weight of all `Http2Stream` instances that depend on this `Http2Stream` as specified using `PRIORITY` frames.
  - `weight` <number> The priority weight of this `Http2Stream`.

A current state of this `Http2Stream`.

Class: ClientHttp2Stream

Added in: v8.4.0

- Extends <Http2Stream>

The `ClientHttp2Stream` class is an extension of `Http2Stream` that is used exclusively on HTTP/2 Clients. `Http2Stream` instances on the client provide events such as `'response'` and `'push'` that are only relevant on the client.

Event: 'continue'

Added in: v8.5.0

Emitted when the server sends a `100 Continue` status, usually because the request contained `Expect: 100-continue`. This is an instruction that the client should send the request body.

Event: 'headers'

Added in: v8.4.0

The `'headers'` event is emitted when an additional block of headers is received for a stream, such as when a block of `1xx` informational headers is received. The listener callback is passed the HTTP/2 Headers Object and flags associated with the headers.

```
stream.on('headers', (headers, flags) => {
  console.log(headers);
});
```

Event: 'push'

Added in: v8.4.0

The `'push'` event is emitted when response headers for a Server Push stream are received. The listener callback is passed the HTTP/2 Headers Object and flags associated with the headers.

```
stream.on('push', (headers, flags) => {
  console.log(headers);
});
```

Event: 'response'

Added in: v8.4.0

The `'response'` event is emitted when a response `HEADERS` frame has been received for this stream from the connected HTTP/2 server. The listener is invoked with two arguments: an Object containing the received HTTP/2 Headers Object, and flags associated with the headers.

```
const http2 = require('http2');
const client = http2.connect('https://localhost');
const req = client.request({ ':path': '/' });
req.on('response', (headers, flags) => {
  console.log(headers[':status']);
});
```

Class: ServerHttp2Stream

Added in: v8.4.0

- Extends: <Http2Stream>

The `ServerHttp2Stream` class is an extension of Http2Stream that is used exclusively on HTTP/2 Servers. `Http2Stream` instances on the server

provide additional methods such as `http2stream.pushStream()` and `http2stream.respond()` that are only relevant on the server.

http2stream.additionalHeaders(headers)

Added in: v8.4.0

- `headers` <HTTP/2 Headers Object>

Sends an additional informational `HEADERS` frame to the connected HTTP/2 peer.

http2stream.headersSent

Added in: v8.4.0

- Value: <boolean>

Boolean (read-only). True if headers were sent, false otherwise.

http2stream.pushAllowed

Added in: v8.4.0

- Value: <boolean>

Read-only property mapped to the `SETTINGS_ENABLE_PUSH` flag of the remote client's most recent `SETTINGS` frame. Will be `true` if the remote peer accepts push streams, `false` otherwise. Settings are the same for every `Http2Stream` in the same `Http2Session`.

http2stream.pushStream(headers[, options], callback)

Added in: v8.4.0

- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `exclusive` <boolean> When `true` and `parent` identifies a parent Stream, the created stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of the newly created stream. **Default:** `false`
  - `parent` <number> Specifies the numeric identifier of a stream the newly created stream is dependent on.
- `callback` <Function> Callback that is called once the push stream has been initiated.
  - `err` <Error>
  - `pushStream` <ServerHttp2Stream> The returned pushStream object.
  - `headers` <HTTP/2 Headers Object> Headers object the pushStream was initiated with.

Initiates a push stream. The callback is invoked with the new `Http2Stream` instance created for the push stream passed as the second argument, or an `Error` passed as the first argument.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.pushStream({ ':path': '/' }, (err, pushStream, headers) => {
    if (err) throw err;
    pushStream.respond({ ':status': 200 });
    pushStream.end('some pushed data');
  });
  stream.end('some data');
});
```

Setting the weight of a push stream is not allowed in the `HEADERS` frame. Pass a `weight` value to `http2stream.priority` with the `silent` option set to `true` to enable server-side bandwidth balancing between concurrent streams.

http2stream.respond([headers[, options]])

Added in: v8.4.0

- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `endStream` <boolean> Set to `true` to indicate that the response will not include payload data.
  - `getTrailers` <Function> Callback function invoked to collect trailer headers.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.end('some data');
});
```

When set, the `options.getTrailers()` function is called immediately after queuing the last chunk of payload data to be sent. The callback is passed a single object (with a `null` prototype) that the listener may use to specify the trailing header fields to send to the peer.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 }, {
    getTrailers(trailers) {
      trailers.ABC = 'some value to send';
    }
  });
  stream.end('some data');
});
```

The HTTP/1 specification forbids trailers from containing HTTP/2 pseudo-header fields (e.g. `':status'`, `':path'`, etc). An `'error'` event will be emitted if the `getTrailers` callback attempts to set such header fields.

http2stream.respondWithFD(fd[, headers[, options]])

Added in: v8.4.0

- `fd` <number> A readable file descriptor.
- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `statCheck` <Function>
  - `getTrailers` <Function> Callback function invoked to collect trailer headers.
  - `offset` <number> The offset position at which to begin reading.
  - `length` <number> The amount of data from the fd to send.

Initiates a response whose data is read from the given file descriptor. No validation is performed on the given file descriptor. If an error occurs while attempting to read data using the file descriptor, the `Http2Stream` will be closed using an `RST_STREAM` frame using the standard `INTERNAL_ERROR` code.

When used, the `Http2Stream` object's Duplex interface will be closed automatically.

```
const http2 = require('http2');
const fs = require('fs');

const server = http2.createServer();
server.on('stream', (stream) => {
  const fd = fs.openSync('/some/file', 'r');

  const stat = fs.fstatSync(fd);
  const headers = {
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain'
  };
  stream.respondWithFD(fd, headers);
  stream.on('close', () => fs.closeSync(fd));
});
```

The optional `options.statCheck` function may be specified to give user code an opportunity to set additional content headers based on the `fs.Stat` details of the given fd. If the `statCheck` function is provided, the `http2stream.respondWithFD()` method will perform an `fs.fstat()` call to collect details on the provided file descriptor.

The `offset` and `length` options may be used to limit the response to a specific range subset. This can be used, for instance, to support HTTP Range requests.

The file descriptor is not closed when the stream is closed, so it will need to be closed manually once it is no longer needed. Note that using the same file descriptor concurrently for multiple streams is not supported and may result in data loss. Re-using a file descriptor after a stream has finished is supported.

When set, the `options.getTrailers()` function is called immediately after queuing the last chunk of payload data to be sent. The callback is passed a single object (with a `null` prototype) that the listener may use to specify the trailing header fields to send to the peer.

```
const http2 = require('http2');
const fs = require('fs');

const server = http2.createServer();
server.on('stream', (stream) => {
  const fd = fs.openSync('/some/file', 'r');

  const stat = fs.fstatSync(fd);
  const headers = {
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain'
  };
  stream.respondWithFD(fd, headers, {
    getTrailers(trailers) {
      trailers.ABC = 'some value to send';
    }
  });

  stream.on('close', () => fs.closeSync(fd));
});
```

The HTTP/1 specification forbids trailers from containing HTTP/2 pseudo-header fields (e.g. `':status'`, `':path'`, etc). An `'error'` event will be emitted if the `getTrailers` callback attempts to set such header fields.

http2stream.respondWithFile(path[, headers[, options]])

Added in: v8.4.0

- `path` <string> | <Buffer> | <URL>
- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `statCheck` <Function>
  - `onError` <Function> Callback function invoked in the case of an Error before send.
  - `getTrailers` <Function> Callback function invoked to collect trailer headers.
  - `offset` <number> The offset position at which to begin reading.
  - `length` <number> The amount of data from the fd to send.

Sends a regular file as the response. The `path` must specify a regular file or an `'error'` event will be emitted on the `Http2Stream` object.

When used, the `Http2Stream` object's Duplex interface will be closed automatically.

The optional `options.statCheck` function may be specified to give user code an opportunity to set additional content headers based on the `fs.Stat` details of the given file:

If an error occurs while attempting to read the file data, the `Http2Stream` will be closed using an `RST_STREAM` frame using the standard `INTERNAL_ERROR` code. If the `onError` callback is defined, then it will be called. Otherwise the stream will be destroyed.

Example using a file path:

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  function statCheck(stat, headers) {
    headers['last-modified'] = stat.mtime.toUTCString();
  }

  function onError(err) {
    if (err.code === 'ENOENT') {
      stream.respond({ ':status': 404 });
    } else {
      stream.respond({ ':status': 500 });
    }
    stream.end();
  }

  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain' },
                         { statCheck, onError });
});
```

The `options.statCheck` function may also be used to cancel the send operation by returning `false`. For instance, a conditional request may check the stat results to determine if the file has been modified to return an appropriate `304` response:

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  function statCheck(stat, headers) {
    // Check the stat here...
    stream.respond({ ':status': 304 });
    return false; // Cancel the send operation
  }
  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain' },
                         { statCheck });
});
```

The `content-length` header field will be automatically set.

The `offset` and `length` options may be used to limit the response to a specific range subset. This can be used, for instance, to support HTTP Range requests.

The `options.onError` function may also be used to handle all the errors that could happen before the delivery of the file is initiated. The default behavior is to destroy the stream.

When set, the `options.getTrailers()` function is called immediately after queuing the last chunk of payload data to be sent. The callback is passed a single object (with a `null` prototype) that the listener may use to specify the trailing header fields to send to the peer.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  function getTrailers(trailers) {
    trailers.ABC = 'some value to send';
  }
  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain' },
                         { getTrailers });
});
```

The HTTP/1 specification forbids trailers from containing HTTP/2 pseudo-header fields (e.g. `':status'`, `':path'`, etc). An `'error'` event will be emitted if the `getTrailers` callback attempts to set such header fields.

Class: Http2Server

Added in: v8.4.0

- Extends: <net.Server>

In `Http2Server`, there are no `'clientError'` events as there are in HTTP1. However, there are `'sessionError'`, and `'streamError'` events for errors emitted on the socket, or from `Http2Session` or `Http2Stream` instances.

Event: 'checkContinue'

Added in: v8.5.0

- `request` <http2.Http2ServerRequest>
- `response` <http2.Http2ServerResponse>

If a `'request'` listener is registered or `http2.createServer()` is supplied a callback function, the `'checkContinue'` event is emitted each time a request with an HTTP `Expect: 100-continue` is received. If this event is not listened for, the server will automatically respond with a status `100 Continue` as appropriate.

Handling this event involves calling `response.writeContinue()` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g. 400 Bad Request) if the client should not continue to send the request body.

Note that when this event is emitted and handled, the 'request' event will not be emitted.

Event: 'request'

Added in: v8.4.0

- request <http2.Http2ServerRequest>
- response <http2.Http2ServerResponse>

Emitted each time there is a request. Note that there may be multiple requests per session. See the Compatibility API .

Event: 'session'

Added in: v8.4.0

The 'session' event is emitted when a new Http2Session is created by the Http2Server .

Event: 'sessionError'

Added in: v8.4.0

The 'sessionError' event is emitted when an 'error' event is emitted by an Http2Session object associated with the Http2Server .

Event: 'streamError'

Added in: v8.5.0

If a ServerHttp2Stream emits an 'error' event, it will be forwarded here. The stream will already be destroyed when this event is triggered.

Event: 'stream'

Added in: v8.4.0

The 'stream' event is emitted when a 'stream' event has been emitted by an Http2Session associated with the server.

```
const http2 = require('http2');
const {
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE
} = http2.constants;

const server = http2.createServer();
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain'
  });
  stream.write('hello ');
  stream.end('world');
});
```

Event: 'timeout'

Added in: v8.4.0

The 'timeout' event is emitted when there is no activity on the Server for a given number of milliseconds set using http2server.setTimeout() .

Class: Http2SecureServer

Added in: v8.4.0

- Extends: <tls.Server>

Event: 'checkContinue'

Added in: v8.5.0

- `request` <http2.Http2ServerRequest>
- `response` <http2.Http2ServerResponse>

If a `'request'` listener is registered or `http2.createSecureServer()` is supplied a callback function, the `'checkContinue'` event is emitted each time a request with an HTTP `Expect: 100-continue` is received. If this event is not listened for, the server will automatically respond with a status `100 Continue` as appropriate.

Handling this event involves calling `response.writeContinue()` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g. 400 Bad Request) if the client should not continue to send the request body.

Note that when this event is emitted and handled, the `'request'` event will not be emitted.

Event: 'request'

Added in: v8.4.0

- `request` <http2.Http2ServerRequest>
- `response` <http2.Http2ServerResponse>

Emitted each time there is a request. Note that there may be multiple requests per session. See the Compatibility API .

Event: 'session'

Added in: v8.4.0

The `'session'` event is emitted when a new `Http2Session` is created by the `Http2SecureServer`.

Event: 'sessionError'

Added in: v8.4.0

The `'sessionError'` event is emitted when an `'error'` event is emitted by an `Http2Session` object associated with the `Http2SecureServer`.

Event: 'stream'

Added in: v8.4.0

The `'stream'` event is emitted when a `'stream'` event has been emitted by an `Http2Session` associated with the server.

```
const http2 = require('http2');
const {
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE
} = http2.constants;

const options = getOptionsSomehow();

const server = http2.createSecureServer(options);
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain'
  });
  stream.write('hello ');
  stream.end('world');
});
```

Event: 'timeout'

Added in: v8.4.0

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2secureServer.setTimeout()` .

Event: 'unknownProtocol'

Added in: v8.4.0

The `'unknownProtocol'` event is emitted when a connecting client fails to negotiate an allowed protocol (i.e. HTTP/2 or HTTP/1.1). The event handler receives the socket for handling. If no listener is registered for this event, the connection is terminated. See the Compatibility API .

http2.createServer(options[, onRequestHandler])

▸ History

- `options` <Object>
  - `maxDeflateDynamicTableSize` <number> Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib`
  - `maxSessionMemory` <number> Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1`. **Default:** `10`. This is a credit based limit, existing `Http2Stream`s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged PING and SETTINGS frames are all counted towards the current limit.
  - `maxHeaderListPairs` <number> Sets the maximum number of header entries. **Default:** `128`. The minimum value is `4`.
  - `maxOutstandingPings` <number> Sets the maximum number of outstanding, unacknowledged pings. The default is `10`.
  - `maxSendHeaderBlockLength` <number> Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
  - `paddingStrategy` <number> Identifies the strategy used for determining the amount of padding to use for HEADERS and DATA frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE`. Value may be one of:
    - `http2.constants.PADDING_STRATEGY_NONE` - Specifies that no padding is to be applied.
    - `http2.constants.PADDING_STRATEGY_MAX` - Specifies that the maximum amount of padding, as determined by the internal implementation, is to be applied.

        - `http2.constants.PADDING_STRATEGY_CALLBACK` - Specifies that the user provided `options.selectPadding` callback is to be used to determine the amount of padding.
        - `http2.constants.PADDING_STRATEGY_ALIGNED` - Will *attempt* to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, however, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum will be used and the total frame length will *not* necessarily be aligned at 8 bytes.
    - `peerMaxConcurrentStreams` <number> Sets the maximum number of concurrent streams for the remote peer as if a SETTINGS frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`
    - `selectPadding` <Function> When `options.paddingStrategy` is equal to `http2.constants.PADDING_STRATEGY_CALLBACK`, provides the callback function used to determine the padding. See Using options.selectPadding.
    - `settings` <HTTP/2 Settings Object> The initial settings to send to the remote peer upon connection.
    - `Http1IncomingMessage` <http.IncomingMessage> Specifies the IncomingMessage class to used for HTTP/1 fallback. Useful for extending the original `http.IncomingMessage`. **Default:** `http.IncomingMessage`
    - `Http1ServerResponse` <http.ServerResponse> Specifies the ServerResponse class to used for HTTP/1 fallback. Useful for extending the original `http.ServerResponse`. **Default:** `http.ServerResponse`
    - `Http2ServerRequest` <http2.Http2ServerRequest> Specifies the Http2ServerRequest class to use. Useful for extending the original `Http2ServerRequest`. **Default:** `Http2ServerRequest`
    - `Http2ServerResponse` <http2.Http2ServerResponse> Specifies the Http2ServerResponse class to use. Useful for extending the original `Http2ServerResponse`. **Default:** `Http2ServerResponse`
- `onRequestHandler` <Function> See Compatibility API
- Returns: <Http2Server>

Returns a `net.Server` instance that creates and manages `Http2Session` instances.

Since there are no browsers known that support unencrypted HTTP/2, the use of `http2.createSecureServer()` is necessary when communicating with browser clients.

```
const http2 = require('http2');

// Create an unencrypted HTTP/2 server.
// Since there are no browsers known that support
// unencrypted HTTP/2, the use of `http2.createSecureServer()`
// is necessary when communicating with browser clients.
const server = http2.createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html',
    ':status': 200
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(80);
```

http2.createSecureServer(options[, onRequestHandler])

▶ History

- `options` <Object>
    - `allowHTTP1` <boolean> Incoming client connections that do not support HTTP/2 will be downgraded to HTTP/1.x when set to `true`. **Default:** `false`. See the `'unknownProtocol'` event. See ALPN negotiation.
    - `maxDeflateDynamicTableSize` <number> Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib`
    - `maxSessionMemory` <number> Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1`. **Default:** `10`. This is a credit based limit, existing

`Http2Stream` s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged PING and SETTINGS frames are all counted towards the current limit.

  - `maxHeaderListPairs` <number> Sets the maximum number of header entries. **Default:** `128`. The minimum value is `4`.
  - `maxOutstandingPings` <number> Sets the maximum number of outstanding, unacknowledged pings. The default is `10`.
  - `maxSendHeaderBlockLength` <number> Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
  - `paddingStrategy` <number> Identifies the strategy used for determining the amount of padding to use for HEADERS and DATA frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE`. Value may be one of:
    - `http2.constants.PADDING_STRATEGY_NONE` - Specifies that no padding is to be applied.
    - `http2.constants.PADDING_STRATEGY_MAX` - Specifies that the maximum amount of padding, as determined by the internal implementation, is to be applied.
    - `http2.constants.PADDING_STRATEGY_CALLBACK` - Specifies that the user provided `options.selectPadding` callback is to be used to determine the amount of padding.
    - `http2.constants.PADDING_STRATEGY_ALIGNED` - Will *attempt* to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, however, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum will be used and the total frame length will *not* necessarily be aligned at 8 bytes.
  - `peerMaxConcurrentStreams` <number> Sets the maximum number of concurrent streams for the remote peer as if a SETTINGS frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`
  - `selectPadding` <Function> When `options.paddingStrategy` is equal to `http2.constants.PADDING_STRATEGY_CALLBACK`, provides the callback function used to determine the padding. See Using options.selectPadding.
  - `settings` <HTTP/2 Settings Object> The initial settings to send to the remote peer upon connection.
  - ...: Any tls.createServer() options can be provided. For servers, the identity options (`pfx` or `key`/`cert`) are usually required.
- `onRequestHandler` <Function> See Compatibility API
- Returns: <Http2SecureServer>

Returns a `tls.Server` instance that creates and manages `Http2Session` instances.

```
const http2 = require('http2');

const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem')
};

// Create a secure HTTP/2 server
const server = http2.createSecureServer(options);

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html',
    ':status': 200
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(80);
```

http2.connect(authority[, options][, listener])

▶ History

- `authority` <string> | <URL>
- `options` <Object>

- `maxDeflateDynamicTableSize` <number> Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib`
- `maxSessionMemory` <number> Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1`. **Default:** `10`. This is a credit based limit, existing `Http2Stream`s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged PING and SETTINGS frames are all counted towards the current limit.
- `maxHeaderListPairs` <number> Sets the maximum number of header entries. **Default:** `128`. The minimum value is `1`.
- `maxOutstandingPings` <number> Sets the maximum number of outstanding, unacknowledged pings. The default is `10`.
- `maxReservedRemoteStreams` <number> Sets the maximum number of reserved push streams the client will accept at any given time. Once the current number of currently reserved push streams exceeds reaches this limit, new push streams sent by the server will be automatically rejected.
- `maxSendHeaderBlockLength` <number> Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
- `paddingStrategy` <number> Identifies the strategy used for determining the amount of padding to use for HEADERS and DATA frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE`. Value may be one of:
  - `http2.constants.PADDING_STRATEGY_NONE` - Specifies that no padding is to be applied.
  - `http2.constants.PADDING_STRATEGY_MAX` - Specifies that the maximum amount of padding, as determined by the internal implementation, is to be applied.
  - `http2.constants.PADDING_STRATEGY_CALLBACK` - Specifies that the user provided `options.selectPadding` callback is to be used to determine the amount of padding.
  - `http2.constants.PADDING_STRATEGY_ALIGNED` - Will *attempt* to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, however, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum will be used and the total frame length will *not* necessarily be aligned at 8 bytes.
- `peerMaxConcurrentStreams` <number> Sets the maximum number of concurrent streams for the remote peer as if a SETTINGS frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`
- `selectPadding` <Function> When `options.paddingStrategy` is equal to `http2.constants.PADDING_STRATEGY_CALLBACK`, provides the callback function used to determine the padding. See Using options.selectPadding.
- `settings` <HTTP/2 Settings Object> The initial settings to send to the remote peer upon connection.
- `createConnection` <Function> An optional callback that receives the `URL` instance passed to `connect` and the `options` object, and returns any Duplex stream that is to be used as the connection for this session.
- ...: Any net.connect() or tls.connect() options can be provided.

- `listener` <Function>
- Returns: <ClientHttp2Session>

Returns a `ClientHttp2Session` instance.

```
const http2 = require('http2');
const client = http2.connect('https://localhost:1234');

/** use the client **/

client.close();
```

http2.constants

Added in: v8.4.0

Error Codes for RST_STREAM and GOAWAY

Value	Name	Constant
0x00	No Error	`http2.constants.NGHTTP2_NO_ERROR`

Value	Name	Constant
0x01	Protocol Error	`http2.constants.NGHTTP2_PROTOCOL_ERROR`
0x02	Internal Error	`http2.constants.NGHTTP2_INTERNAL_ERROR`
0x03	Flow Control Error	`http2.constants.NGHTTP2_FLOW_CONTROL_ERROR`
0x04	Settings Timeout	`http2.constants.NGHTTP2_SETTINGS_TIMEOUT`
0x05	Stream Closed	`http2.constants.NGHTTP2_STREAM_CLOSED`
0x06	Frame Size Error	`http2.constants.NGHTTP2_FRAME_SIZE_ERROR`
0x07	Refused Stream	`http2.constants.NGHTTP2_REFUSED_STREAM`
0x08	Cancel	`http2.constants.NGHTTP2_CANCEL`
0x09	Compression Error	`http2.constants.NGHTTP2_COMPRESSION_ERROR`
0x0a	Connect Error	`http2.constants.NGHTTP2_CONNECT_ERROR`
0x0b	Enhance Your Calm	`http2.constants.NGHTTP2_ENHANCE_YOUR_CALM`
0x0c	Inadequate Security	`http2.constants.NGHTTP2_INADEQUATE_SECURITY`
0x0d	HTTP/1.1 Required	`http2.constants.NGHTTP2_HTTP_1_1_REQUIRED`

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2server.setTimeout()`.

http2.getDefaultSettings()

Added in: v8.4.0

- Returns: <HTTP/2 Settings Object>

Returns an object containing the default settings for an `Http2Session` instance. This method returns a new object instance every time it is called so instances returned may be safely modified for use.

http2.getPackedSettings(settings)

Added in: v8.4.0

- `settings` <HTTP/2 Settings Object>
- Returns: <Buffer>

Returns a `Buffer` instance containing serialized representation of the given HTTP/2 settings as specified in the HTTP/2 specification. This is intended for use with the `HTTP2-Settings` header field.

```
const http2 = require('http2');

const packed = http2.getPackedSettings({ enablePush: false });

console.log(packed.toString('base64'));
// Prints: AAIAAAAA
```

http2.getUnpackedSettings(buf)

Added in: v8.4.0

- `buf` <Buffer> | <Uint8Array> The packed settings.

- Returns: <HTTP/2 Settings Object>

Returns a HTTP/2 Settings Object containing the deserialized settings from the given `Buffer` as generated by `http2.getPackedSettings()`.

Headers Object

Headers are represented as own-properties on JavaScript objects. The property keys will be serialized to lower-case. Property values should be strings (if they are not they will be coerced to strings) or an Array of strings (in order to send more than one value per header field).

```
const headers = {
  ':status': '200',
  'content-type': 'text-plain',
  'ABC': ['has', 'more', 'than', 'one', 'value']
};

stream.respond(headers);
```

Header objects passed to callback functions will have a `null` prototype. This means that normal JavaScript object methods such as `Object.prototype.toString()` and `Object.prototype.hasOwnProperty()` will not work.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream, headers) => {
  console.log(headers[':path']);
  console.log(headers.ABC);
});
```

Settings Object

▸ History

The `http2.getDefaultSettings()`, `http2.getPackedSettings()`, `http2.createServer()`, `http2.createSecureServer()`, `http2session.settings()`, `http2session.localSettings`, and `http2session.remoteSettings` APIs either return or receive as input an object that defines configuration settings for an `Http2Session` object. These objects are ordinary JavaScript objects containing the following properties.

- `headerTableSize` <number> Specifies the maximum number of bytes used for header compression. **Default:** `4,096 octets`. The minimum allowed value is 0. The maximum allowed value is 2^{32}-1.
- `enablePush` <boolean> Specifies `true` if HTTP/2 Push Streams are to be permitted on the `Http2Session` instances.
- `initialWindowSize` <number> Specifies the *senders* initial window size for stream-level flow control. **Default:** `65,535 bytes`. The minimum allowed value is 0. The maximum allowed value is 2^{32}-1.
- `maxFrameSize` <number> Specifies the size of the largest frame payload. **Default:** `16,384 bytes`. The minimum allowed value is 16,384. The maximum allowed value is 2^{24}-1.
- `maxConcurrentStreams` <number> Specifies the maximum number of concurrent streams permitted on an `Http2Session`. There is no default value which implies, at least theoretically, 2^{31}-1 streams may be open concurrently at any given time in an `Http2Session`. The minimum value is 0. The maximum allowed value is 2^{31}-1.
- `maxHeaderListSize` <number> Specifies the maximum size (uncompressed octets) of header list that will be accepted. The minimum allowed value is 0. The maximum allowed value is 2^{32}-1. **Default:** 65535.

All additional properties on the settings object are ignored.

Using `options.selectPadding`

When `options.paddingStrategy` is equal to `http2.constants.PADDING_STRATEGY_CALLBACK`, the HTTP/2 implementation will consult the `options.selectPadding` callback function, if provided, to determine the specific amount of padding to use per HEADERS and DATA frame.

The `options.selectPadding` function receives two numeric arguments, `frameLen` and `maxFrameLen` and must return a number `N` such that `frameLen <= N <= maxFrameLen`.

```
const http2 = require('http2');
const server = http2.createServer({
  paddingStrategy: http2.constants.PADDING_STRATEGY_CALLBACK,
  selectPadding(frameLen, maxFrameLen) {
    return maxFrameLen;
  }
});
```

The `options.selectPadding` function is invoked once for *every* HEADERS and DATA frame. This has a definite noticeable impact on performance.

Error Handling

There are several types of error conditions that may arise when using the `http2` module:

Validation Errors occur when an incorrect argument, option, or setting value is passed in. These will always be reported by a synchronous `throw`.

State Errors occur when an action is attempted at an incorrect time (for instance, attempting to send data on a stream after it has closed). These will be reported using either a synchronous `throw` or via an `'error'` event on the `Http2Stream`, `Http2Session` or HTTP/2 Server objects, depending on where and when the error occurs.

Internal Errors occur when an HTTP/2 session fails unexpectedly. These will be reported via an `'error'` event on the `Http2Session` or HTTP/2 Server objects.

Protocol Errors occur when various HTTP/2 protocol constraints are violated. These will be reported using either a synchronous `throw` or via an `'error'` event on the `Http2Stream`, `Http2Session` or HTTP/2 Server objects, depending on where and when the error occurs.

Invalid character handling in header names and values

The HTTP/2 implementation applies stricter handling of invalid characters in HTTP header names and values than the HTTP/1 implementation.

Header field names are *case-insensitive* and are transmitted over the wire strictly as lower-case strings. The API provided by Node.js allows header names to be set as mixed-case strings (e.g. `Content-Type`) but will convert those to lower-case (e.g. `content-type`) upon transmission.

Header field-names *must only* contain one or more of the following ASCII characters: `a`-`z`, `A`-`Z`, `0`-`9`, `!`, `#`, `$`, `%`, `&`, `'`, `*`, `+`, `-`, `.`, `^`, `_`, `` ` `` (backtick), `|`, and `~`.

Using invalid characters within an HTTP header field name will cause the stream to be closed with a protocol error being reported.

Header field values are handled with more leniency but *should* not contain new-line or carriage return characters and *should* be limited to US-ASCII characters, per the requirements of the HTTP specification.

Push streams on the client

To receive pushed streams on the client, set a listener for the `'stream'` event on the `ClientHttp2Session`:

```
const http2 = require('http2');

const client = http2.connect('http://localhost');

client.on('stream', (pushedStream, requestHeaders) => {
  pushedStream.on('push', (responseHeaders) => {
    // process response headers
  });
  pushedStream.on('data', (chunk) => { /* handle pushed data */ });
});

const req = client.request({ ':path': '/' });
```

Supporting the CONNECT method

The `CONNECT` method is used to allow an HTTP/2 server to be used as a proxy for TCP/IP connections.

A simple TCP Server:

```
const net = require('net');

const server = net.createServer((socket) => {
  let name = '';
  socket.setEncoding('utf8');
  socket.on('data', (chunk) => name += chunk);
  socket.on('end', () => socket.end(`hello ${name}`));
});

server.listen(8000);
```

An HTTP/2 CONNECT proxy:

```
const http2 = require('http2');
const { NGHTTP2_REFUSED_STREAM } = http2.constants;
const net = require('net');
const { URL } = require('url');

const proxy = http2.createServer();
proxy.on('stream', (stream, headers) => {
  if (headers[':method'] !== 'CONNECT') {
    // Only accept CONNECT requests
    stream.close(NGHTTP2_REFUSED_STREAM);
    return;
  }
  const auth = new URL(`tcp://${headers[':authority']}`);
  // It's a very good idea to verify that hostname and port are
  // things this proxy should be connecting to.
  const socket = net.connect(auth.port, auth.hostname, () => {
    stream.respond();
    socket.pipe(stream);
    stream.pipe(socket);
  });
  socket.on('error', (error) => {
    stream.close(http2.constants.NGHTTP2_CONNECT_ERROR);
  });
});

proxy.listen(8001);
```

An HTTP/2 CONNECT client:

```
const http2 = require('http2');

const client = http2.connect('http://localhost:8001');

// Must not specify the ':path' and ':scheme' headers
// for CONNECT requests or an error will be thrown.
const req = client.request({
  ':method': 'CONNECT',
  ':authority': `localhost:${port}`
});

req.on('response', (headers) => {
  console.log(headers[http2.constants.HTTP2_HEADER_STATUS]);
});
let data = '';
req.setEncoding('utf8');
req.on('data', (chunk) => data += chunk);
req.on('end', () => {
  console.log(`The server says: ${data}`);
  client.close();
});
req.end('Jane');
```

Compatibility API

The Compatibility API has the goal of providing a similar developer experience of HTTP/1 when using HTTP/2, making it possible to develop applications that support both HTTP/1 and HTTP/2. This API targets only the **public API** of the HTTP/1. However many modules use

internal methods or state, and those *are not supported* as it is a completely different implementation.

The following example creates an HTTP/2 server using the compatibility API:

```
const http2 = require('http2');
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('ok');
});
```

In order to create a mixed HTTPS and HTTP/2 server, refer to the ALPN negotiation section. Upgrading from non-tls HTTP/1 servers is not supported.

The HTTP/2 compatibility API is composed of `Http2ServerRequest` and `Http2ServerResponse`. They aim at API compatibility with HTTP/1, but they do not hide the differences between the protocols. As an example, the status message for HTTP codes is ignored.

ALPN negotiation

ALPN negotiation allows supporting both HTTPS and HTTP/2 over the same socket. The `req` and `res` objects can be either HTTP/1 or HTTP/2, and an application **must** restrict itself to the public API of HTTP/1, and detect if it is possible to use the more advanced features of HTTP/2.

The following example creates a server that supports both protocols:

```
const { createSecureServer } = require('http2');
const { readFileSync } = require('fs');

const cert = readFileSync('./cert.pem');
const key = readFileSync('./key.pem');

const server = createSecureServer(
  { cert, key, allowHTTP1: true },
  onRequest
).listen(4443);

function onRequest(req, res) {
  // detects if it is a HTTPS request or HTTP/2
  const { socket: { alpnProtocol } } = req.httpVersion === '2.0' ?
    req.stream.session : req;
  res.writeHead(200, { 'content-type': 'application/json' });
  res.end(JSON.stringify({
    alpnProtocol,
    httpVersion: req.httpVersion
  }));
}
```

The `'request'` event works identically on both HTTPS and HTTP/2.

Class: http2.Http2ServerRequest

Added in: v8.4.0

A `Http2ServerRequest` object is created by `http2.Server` or `http2.SecureServer` and passed as the first argument to the `'request'` event. It may be used to access a request status, headers, and data.

It implements the Readable Stream interface, as well as the following additional events, methods, and properties.

Event: 'aborted'

Added in: v8.4.0

The `'aborted'` event is emitted whenever a `Http2ServerRequest` instance is abnormally aborted in mid-communication.

The `'aborted'` event will only be emitted if the `Http2ServerRequest` writable side has not been ended.

Event: 'close'

Added in: v8.4.0

Indicates that the underlying Http2Stream was closed. Just like `'end'`, this event occurs only once per response.

request.destroy([error])

Added in: v8.4.0

- `error` <Error>

Calls `destroy()` on the Http2Stream that received the Http2ServerRequest. If `error` is provided, an `'error'` event is emitted and `error` is passed as an argument to any listeners on the event.

It does nothing if the stream was already destroyed.

request.headers

Added in: v8.4.0

- <Object>

The request/response headers object.

Key-value pairs of header names and values. Header names are lower-cased. Example:

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

See HTTP/2 Headers Object.

In HTTP/2, the request path, hostname, protocol, and method are represented as special headers prefixed with the `:` character (e.g. `':path'`). These special headers will be included in the `request.headers` object. Care must be taken not to inadvertently modify these special headers or errors may occur. For instance, removing all headers from the request will cause errors to occur:

```
removeAllHeaders(request.headers);
assert(request.url);   // Fails because the :path header has been removed
```

request.httpVersion

Added in: v8.4.0

- <string>

In case of server request, the HTTP version sent by the client. In the case of client response, the HTTP version of the connected-to server. Returns `'2.0'`.

Also `message.httpVersionMajor` is the first integer and `message.httpVersionMinor` is the second.

request.method

Added in: v8.4.0

- <string>

The request method as a string. Read-only. Example: 'GET', 'DELETE'.

request.rawHeaders

Added in: v8.4.0

- <Array>

The raw request/response headers list exactly as they were received.

Note that the keys and values are in the same list. It is *not* a list of tuples. So, the even-numbered offsets are key values, and the odd-numbered offsets are the associated values.

Header names are not lowercased, and duplicates are not merged.

```
// Prints something like:
//
// [ 'user-agent',
//   'this is invalid because there can be only one',
//   'User-Agent',
//   'curl/7.22.0',
//   'Host',
//   '127.0.0.1:8000',
//   'ACCEPT',
//   '*/*' ]
console.log(request.rawHeaders);
```

request.rawTrailers

Added in: v8.4.0

- <Array>

The raw request/response trailer keys and values exactly as they were received. Only populated at the 'end' event.

request.setTimeout(msecs, callback)

Added in: v8.4.0

- msecs <number>
- callback <Function>

Sets the Http2Stream's timeout value to msecs. If a callback is provided, then it is added as a listener on the 'timeout' event on the response object.

If no 'timeout' listener is added to the request, the response, or the server, then Http2Streams are destroyed when they time out. If a handler is assigned to the request, the response, or the server's 'timeout' events, timed out sockets must be handled explicitly.

Returns request.

request.socket

Added in: v8.4.0

- <net.Socket> | <tls.TLSSocket>

Returns a Proxy object that acts as a net.Socket (or tls.TLSSocket) but applies getters, setters, and methods based on HTTP/2 logic.

destroyed, readable, and writable properties will be retrieved from and set on request.stream.

destroy, emit, end, on and once methods will be called on request.stream.

setTimeout method will be called on request.stream.session.

pause, read, resume, and write will throw an error with code ERR_HTTP2_NO_SOCKET_MANIPULATION. See Http2Session and Sockets for more information.

All other interactions will be routed directly to the socket. With TLS support, use request.socket.getPeerCertificate() to obtain the client's authentication details.

request.stream

Added in: v8.4.0

- <http2.Http2Stream>

The Http2Stream object backing the request.

request.trailers

Added in: v8.4.0

- <Object>

The request/response trailers object. Only populated at the 'end' event.

request.url

Added in: v8.4.0

- <string>

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

Then request.url will be:

```
'/status?name=ryan'
```

To parse the url into its parts require('url').parse(request.url) can be used. Example:

```
$ node
> require('url').parse('/status?name=ryan')
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status',
  path: '/status?name=ryan',
  href: '/status?name=ryan' }
```

To extract the parameters from the query string, the require('querystring').parse function can be used, or true can be passed as the second argument to require('url').parse. Example:

```
$ node
> require('url').parse('/status?name=ryan', true)
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=ryan',
  query: { name: 'ryan' },
  pathname: '/status',
  path: '/status?name=ryan',
  href: '/status?name=ryan' }
```

Class: http2.Http2ServerResponse

Added in: v8.4.0

This object is created internally by an HTTP server — not by the user. It is passed as the second parameter to the `'request'` event.

The response implements, but does not inherit from, the Writable Stream interface. This is an `EventEmitter` with the following events:

Event: 'close'

Added in: v8.4.0

Indicates that the underlying `Http2Stream` was terminated before `response.end()` was called or able to flush.

Event: 'finish'

Added in: v8.4.0

Emitted when the response has been sent. More specifically, this event is emitted when the last segment of the response headers and body have been handed off to the HTTP/2 multiplexing for transmission over the network. It does not imply that the client has received anything yet.

After this event, no more events will be emitted on the response object.

response.addTrailers(headers)

Added in: v8.4.0

- `headers` <Object>

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

response.connection

Added in: v8.4.0

- <net.Socket> | <tls.TLSSocket>

See `response.socket`.

response.end([data][, encoding][, callback])

Added in: v8.4.0

- `data` <string> | <Buffer>

- `encoding` <string>
- `callback` <Function>

This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, `response.end()`, MUST be called on each response.

If `data` is specified, it is equivalent to calling `response.write(data, encoding)` followed by `response.end(callback)`.

If `callback` is specified, it will be called when the response stream is finished.

response.finished

Added in: v8.4.0

- <boolean>

Boolean value that indicates whether the response has completed. Starts as `false`. After `response.end()` executes, the value will be `true`.

response.getHeader(name)

Added in: v8.4.0

- `name` <string>
- Returns: <string>

Reads out a header that has already been queued but not sent to the client. Note that the name is case insensitive.

Example:

```
const contentType = response.getHeader('content-type');
```

response.getHeaderNames()

Added in: v8.4.0

- Returns: <Array>

Returns an array containing the unique names of the current outgoing headers. All header names are lowercase.

Example:

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headerNames = response.getHeaderNames();
// headerNames === ['foo', 'set-cookie']
```

response.getHeaders()

Added in: v8.4.0

- Returns: <Object>

Returns a shallow copy of the current outgoing headers. Since a shallow copy is used, array values may be mutated without additional calls to various header-related http module methods. The keys of the returned object are the header names and the values are the respective header values. All header names are lowercase.

The object returned by the `response.getHeaders()` method *does not* prototypically inherit from the JavaScript `Object`. This means that typical `Object` methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and *will not work*.

Example:

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headers = response.getHeaders();
// headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }
```

response.hasHeader(name)

Added in: v8.4.0

- `name` <string>
- Returns: <boolean>

Returns `true` if the header identified by `name` is currently set in the outgoing headers. Note that the header name matching is case-insensitive.

Example:

```
const hasContentType = response.hasHeader('content-type');
```

response.headersSent

Added in: v8.4.0

- <boolean>

Boolean (read-only). True if headers were sent, false otherwise.

response.removeHeader(name)

Added in: v8.4.0

- `name` <string>

Removes a header that has been queued for implicit sending.

Example:

```
response.removeHeader('Content-Encoding');
```

response.sendDate

Added in: v8.4.0

- <boolean>

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

response.setHeader(name, value)

Added in: v8.4.0

- `name` <string>
- `value` <string> | <string[]>

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here to send multiple headers with the same name.

Example:

```
response.setHeader('Content-Type', 'text/html');
```

or

```
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

Attempting to set a header field name or value that contains invalid characters will result in a TypeError being thrown.

When headers have been set with response.setHeader(), they will be merged with any headers passed to response.writeHead(), with the headers passed to response.writeHead() given precedence.

```
// returns content-type = text/plain
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('ok');
});
```

response.setTimeout(msecs[, callback])

Added in: v8.4.0

- `msecs` <number>
- `callback` <Function>

Sets the Http2Stream's timeout value to `msecs`. If a callback is provided, then it is added as a listener on the `'timeout'` event on the response object.

If no `'timeout'` listener is added to the request, the response, or the server, then Http2Stream s are destroyed when they time out. If a handler is assigned to the request, the response, or the server's `'timeout'` events, timed out sockets must be handled explicitly.

Returns `response`.

response.socket

Added in: v8.4.0

- <net.Socket> | <tls.TLSSocket>

Returns a Proxy object that acts as a `net.Socket` (or `tls.TLSSocket`) but applies getters, setters, and methods based on HTTP/2 logic.

`destroyed`, `readable`, and `writable` properties will be retrieved from and set on `response.stream`.

`destroy`, `emit`, `end`, `on` and `once` methods will be called on `response.stream`.

`setTimeout` method will be called on `response.stream.session`.

`pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See Http2Session and Sockets for more information.

All other interactions will be routed directly to the socket.

Example:

```
const http2 = require('http2');
const server = http2.createServer((req, res) => {
  const ip = req.socket.remoteAddress;
  const port = req.socket.remotePort;
  res.end(`Your IP address is ${ip} and your source port is ${port}.`);
}).listen(3000);
```

response.statusCode

Added in: v8.4.0

- <number>

When using implicit headers (not calling response.writeHead() explicitly), this property controls the status code that will be sent to the client when the headers get flushed.

Example:

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code which was sent out.

response.statusMessage

Added in: v8.4.0

- <string>

Status message is not supported by HTTP/2 (RFC7540 8.1.2.4). It returns an empty string.

response.stream

Added in: v8.4.0

- <http2.Http2Stream>

The Http2Stream object backing the response.

response.write(chunk[, encoding][, callback])

Added in: v8.4.0

- `chunk` <string> | <Buffer>
- `encoding` <string>
- `callback` <Function>
- Returns: <boolean>

If this method is called and response.writeHead() has not been called, it will switch to implicit header mode and flush the implicit headers.

This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

Note that in the `http` module, the response body is omitted when the request is a HEAD request. Similarly, the `204` and `304` responses *must not* include a message body.

`chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. By default the `encoding` is `'utf8'`. `callback` will be called when this chunk of data is flushed.

This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time response.write() is called, it will send the buffered header information and the first chunk of the body to the client. The second time response.write() is called, Node.js assumes data will be streamed, and sends the new data separately. That is, the response is buffered up to the first chunk of the body.

Returns true if the entire data was flushed successfully to the kernel buffer. Returns false if all or part of the data was queued in user memory. 'drain' will be emitted when the buffer is free again.

response.writeContinue()

Added in: v8.4.0

Sends a status 100 Continue to the client, indicating that the request body should be sent. See the 'checkContinue' event on Http2Server and Http2SecureServer.

response.writeHead(statusCode[, statusMessage][, headers])

Added in: v8.4.0

- statusCode <number>
- statusMessage <string>
- headers <Object>

Sends a response header to the request. The status code is a 3-digit HTTP status code, like 404. The last argument, headers, are the response headers.

For compatibility with HTTP/1, a human-readable statusMessage may be passed as the second argument. However, because the statusMessage has no meaning within HTTP/2, the argument will have no effect and a process warning will be emitted.

Example:

```
const body = 'hello world';
response.writeHead(200, {
  'Content-Length': Buffer.byteLength(body),
  'Content-Type': 'text/plain' });
```

Note that Content-Length is given in bytes not characters. The Buffer.byteLength() API may be used to determine the number of bytes in a given encoding. On outbound messages, Node.js does not check if Content-Length and the length of the body being transmitted are equal or not. However, when receiving messages, Node.js will automatically reject messages when the Content-Length does not match the actual payload size.

This method may be called at most one time on a message before response.end() is called.

If response.write() or response.end() are called before calling this, the implicit/mutable headers will be calculated and call this function.

When headers have been set with response.setHeader(), they will be merged with any headers passed to response.writeHead(), with the headers passed to response.writeHead() given precedence.

```
// returns content-type = text/plain
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('ok');
});
```

Attempting to set a header field name or value that contains invalid characters will result in a TypeError being thrown.

response.createPushResponse(headers, callback)

Added in: v8.4.0

Call http2stream.pushStream() with the given headers, and wraps the given newly created Http2Stream on Http2ServerRespose.

The callback will be called with an error with code ERR_HTTP2_STREAM_CLOSED if the stream is closed.

Collecting HTTP/2 Performance Metrics

The Performance Observer API can be used to collect basic performance metrics for each `Http2Session` and `Http2Stream` instance.

```
const { PerformanceObserver } = require('perf_hooks');

const obs = new PerformanceObserver((items) => {
  const entry = items.getEntries()[0];
  console.log(entry.entryType);  // prints 'http2'
  if (entry.name === 'Http2Session') {
    // entry contains statistics about the Http2Session
  } else if (entry.name === 'Http2Stream') {
    // entry contains statistics about the Http2Stream
  }
});
obs.observe({ entryTypes: ['http2'] });
```

The `entryType` property of the `PerformanceEntry` will be equal to `'http2'`.

The `name` property of the `PerformanceEntry` will be equal to either `'Http2Stream'` or `'Http2Session'`.

If `name` is equal to `Http2Stream`, the `PerformanceEntry` will contain the following additional properties:

- `bytesRead` <number> The number of DATA frame bytes received for this `Http2Stream`.
- `bytesWritten` <number> The number of DATA frame bytes sent for this `Http2Stream`.
- `id` <number> The identifier of the associated `Http2Stream`
- `timeToFirstByte` <number> The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first `DATA` frame.
- `timeToFirstByteSent` <number> The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and sending of the first `DATA` frame.
- `timeToFirstHeader` <number> The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first header.

If `name` is equal to `Http2Session`, the `PerformanceEntry` will contain the following additional properties:

- `bytesRead` <number> The number of bytes received for this `Http2Session`.
- `bytesWritten` <number> The number of bytes sent for this `Http2Session`.
- `framesReceived` <number> The number of HTTP/2 frames received by the `Http2Session`.
- `framesSent` <number> The number of HTTP/2 frames sent by the `Http2Session`.
- `maxConcurrentStreams` <number> The maximum number of streams concurrently open during the lifetime of the `Http2Session`.
- `pingRTT` <number> The number of milliseconds elapsed since the transmission of a `PING` frame and the reception of its acknowledgment. Only present if a `PING` frame has been sent on the `Http2Session`.
- `streamAverageDuration` <number> The average duration (in milliseconds) for all `Http2Stream` instances.
- `streamCount` <number> The number of `Http2Stream` instances processed by the `Http2Session`.
- `type` <string> Either `'server'` or `'client'` to identify the type of `Http2Session`.

HTTPS

Stability: 2 - Stable

HTTPS is the HTTP protocol over TLS/SSL. In Node.js this is implemented as a separate module.

Class: https.Agent

Added in: v0.4.5

An Agent object for HTTPS similar to http.Agent . See https.request() for more information.

Class: https.Server

Added in: v0.3.4

This class is a subclass of tls.Server and emits events same as http.Server . See http.Server for more information.

server.close([callback])

Added in: v0.1.90

- callback <Function>

See server.close() from the HTTP module for details.

server.listen()

Starts the HTTPS server listening for encrypted connections. This method is identical to server.listen() from net.Server .

server.setTimeout([msecs][, callback])

Added in: v0.11.2

- msecs <number> Defaults to 120000 (2 minutes).
- callback <Function>

See http.Server#setTimeout() .

server.timeout

Added in: v0.11.2

- <number> Defaults to 120000 (2 minutes).

See http.Server#timeout .

server.keepAliveTimeout

Added in: v8.0.0

- <number> Defaults to 5000 (5 seconds).

See http.Server#keepAliveTimeout .

https.createServer([options][, requestListener])

Added in: v0.3.4

- options <Object> Accepts options from tls.createServer() , tls.createSecureContext() and http.createServer() .
- requestListener <Function> A listener to be added to the request event.

Example:

```
// curl -k https://localhost:8000/
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Or

```
const https = require('https');
const fs = require('fs');

const options = {
  pfx: fs.readFileSync('test/fixtures/test_cert.pfx'),
  passphrase: 'sample'
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

https.get(options[, callback])

▶ History

- `options` <Object> | <string> | <URL> Accepts the same `options` as https.request(), with the `method` always set to `GET`.
- `callback` <Function>

Like http.get() but for HTTPS.

`options` can be an object, a string, or a URL object. If `options` is a string, it is automatically parsed with url.parse(). If it is a URL object, it will be automatically converted to an ordinary `options` object.

Example:

```
const https = require('https');

https.get('https://encrypted.google.com/', (res) => {
  console.log('statusCode:', res.statusCode);
  console.log('headers:', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });

}).on('error', (e) => {
  console.error(e);
});
```

https.globalAgent

Added in: v0.5.9

Global instance of `https.Agent` for all HTTPS client requests.

https.request(options[, callback])

▶ History

- `options` <Object> | <string> | <URL> Accepts all `options` from `http.request()`, with some differences in default values:
  - `protocol` Defaults to `https:`
  - `port` Defaults to `443`.
  - `agent` Defaults to `https.globalAgent`.
- `callback` <Function>

Makes a request to a secure web server.

The following additional `options` from `tls.connect()` are also accepted: `ca`, `cert`, `ciphers`, `clientCertEngine`, `crl`, `dhparam`, `ecdhCurve`, `honorCipherOrder`, `key`, `passphrase`, `pfx`, `rejectUnauthorized`, `secureOptions`, `secureProtocol`, `servername`, `sessionIdContext`

`options` can be an object, a string, or a `URL` object. If `options` is a string, it is automatically parsed with `url.parse()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

Example:

```
const https = require('https');

const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET'
};

const req = https.request(options, (res) => {
  console.log('statusCode:', res.statusCode);
  console.log('headers:', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });
});

req.on('error', (e) => {
  console.error(e);
});
req.end();
```

Example using options from `tls.connect()` :

```
const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};
options.agent = new https.Agent(options);

const req = https.request(options, (res) => {
  // ...
});
```

Alternatively, opt out of connection pooling by not using an `Agent` .

Example:

```js
const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
  agent: false
};

const req = https.request(options, (res) => {
  // ...
});
```

Example using a `URL` as `options`:

```js
const { URL } = require('url');

const options = new URL('https://abc:xyz@example.com');

const req = https.request(options, (res) => {
  // ...
});
```

Example pinning on certificate fingerprint, or the public key (similar to `pin-sha256`):

```js
const tls = require('tls');
const https = require('https');
const crypto = require('crypto');

function sha256(s) {
  return crypto.createHash('sha256').update(s).digest('base64');
}
const options = {
  hostname: 'github.com',
  port: 443,
  path: '/',
  method: 'GET',
  checkServerIdentity: function(host, cert) {
    // Make sure the certificate is issued to the host we are connected to
    const err = tls.checkServerIdentity(host, cert);
    if (err) {
      return err;
    }

    // Pin the public key, similar to HPKP pin-sha25 pinning
    const pubkey256 = 'pL1+qb9HTMRZJmuC/bB/ZI9d302BYrrqiVuRyW+DGrU=';
    if (sha256(cert.pubkey) !== pubkey256) {
      const msg = 'Certificate verification error: ' +
        `The public key of '${cert.subject.CN}' ` +
        'does not match our pinned fingerprint';
      return new Error(msg);
    }

    // Pin the exact certificate, rather then the pub key
    const cert256 = '25:FE:39:32:D9:63:8C:8A:FC:A1:9A:29:87:' +
```

```
      'D8:3E:4C:1D:98:DB:71:E4:1A:48:03:98:EA:22:6A:BD:8B:93:16';
    if (cert.fingerprint256 !== cert256) {
      const msg = 'Certificate verification error: ' +
        `The certificate of '${cert.subject.CN}' ` +
        'does not match our pinned fingerprint';
      return new Error(msg);
    }

    // This loop is informational only.
    // Print the certificate and public key fingerprints of all certs in the
    // chain. Its common to pin the public key of the issuer on the public
    // internet, while pinning the public key of the service in sensitive
    // environments.
    do {
      console.log('Subject Common Name:', cert.subject.CN);
      console.log('  Certificate SHA256 fingerprint:', cert.fingerprint256);

      hash = crypto.createHash('sha256');
      console.log('  Public key ping-sha256:', sha256(cert.pubkey));

      lastprint256 = cert.fingerprint256;
      cert = cert.issuerCertificate;
    } while (cert.fingerprint256 !== lastprint256);

  },
};

options.agent = new https.Agent(options);
const req = https.request(options, (res) => {
  console.log('All OK. Server matched our pinned cert or public key');
  console.log('statusCode:', res.statusCode);
  // Print the HPKP values
  console.log('headers:', res.headers['public-key-pins']);

  res.on('data', (d) => {});
});

req.on('error', (e) => {
  console.error(e.message);
});
req.end();
```

Outputs for example:

```
Subject Common Name: github.com
  Certificate SHA256 fingerprint: 25:FE:39:32:D9:63:8C:8A:FC:A1:9A:29:87:D8:3E:4C:1D:98:DB:71:E4:1A:48:03:98:EA:22:6A:BD:8B:93:16
  Public key ping-sha256: pL1+qb9HTMRZJmuC/bB/ZI9d302BYrrqiVuRyW+DGrU=
Subject Common Name: DigiCert SHA2 Extended Validation Server CA
  Certificate SHA256 fingerprint: 40:3E:06:2A:26:53:05:91:13:28:5B:AF:80:A0:D4:AE:42:2C:84:8C:9F:78:FA:D0:1F:C9:4B:C5:B8:7F:EF:1A
  Public key ping-sha256: RRM1dGqnDFsCJXBTHky16vi1obOlCgFFn/yOhI/y+ho=
Subject Common Name: DigiCert High Assurance EV Root CA
  Certificate SHA256 fingerprint: 74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3:B1:18:CF
  Public key ping-sha256: WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18=
All OK. Server matched our pinned cert or public key
statusCode: 200
headers: max-age=0; pin-sha256="WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18="; pin-sha256="RRM1dGqnDFsCJXBTHky16vi1obOlCgFFn/yOhI/y+I
```

Inspector

Stability: 1 - Experimental

The inspector module provides an API for interacting with the V8 inspector.

It can be accessed using:

```
const inspector = require('inspector');
```

inspector.open([port[, host[, wait]]])

- port <number> Port to listen on for inspector connections. Optional, defaults to what was specified on the CLI.
- host <string> Host to listen on for inspector connections. Optional, defaults to what was specified on the CLI.
- wait <boolean> Block until a client has connected. Optional, defaults to false.

Activate inspector on host and port. Equivalent to `node --inspect=[[host:]port]` , but can be done programmatically after node has started.

If wait is `true` , will block until a client has connected to the inspect port and flow control has been passed to the debugger client.

inspector.close()

Deactivate the inspector. Blocks until there are no active connections.

inspector.url()

Return the URL of the active inspector, or `undefined` if there is none.

Class: inspector.Session

The `inspector.Session` is used for dispatching messages to the V8 inspector back-end and receiving message responses and notifications.

Constructor: new inspector.Session()

Added in: v8.0.0

Create a new instance of the `inspector.Session` class. The inspector session needs to be connected through session.connect() before the messages can be dispatched to the inspector backend.

`inspector.Session` is an EventEmitter with the following events:

Event: 'inspectorNotification'

Added in: v8.0.0

- <Object> The notification message object

Emitted when any notification from the V8 Inspector is received.

```
session.on('inspectorNotification', (message) => console.log(message.method));
// Debugger.paused
// Debugger.resumed
```

It is also possible to subscribe only to notifications with specific method:

Event: <inspector-protocol-method>

Added in: v8.0.0

- <Object> The notification message object

Emitted when an inspector notification is received that has its method field set to the `<inspector-protocol-method>` value.

The following snippet installs a listener on the `Debugger.paused` event, and prints the reason for program suspension whenever program execution is suspended (through breakpoints, for example):

```
session.on('Debugger.paused', ({ params }) => {
  console.log(params.hitBreakpoints);
});
// [ '/the/file/that/has/the/breakpoint.js:11:0' ]
```

session.connect()

Added in: v8.0.0

Connects a session to the inspector back-end. An exception will be thrown if there is already a connected session established either through the API or by a front-end connected to the Inspector WebSocket port.

session.post(method[, params][, callback])

Added in: v8.0.0

- method <string>
- params <Object>
- callback <Function>

Posts a message to the inspector back-end. `callback` will be notified when a response is received. `callback` is a function that accepts two optional arguments - error and message-specific result.

```
session.post('Runtime.evaluate', { expression: '2 + 2' },
             (error, { result }) => console.log(result));
// Output: { type: 'number', value: 4, description: '4' }
```

The latest version of the V8 inspector protocol is published on the Chrome DevTools Protocol Viewer .

Node inspector supports all the Chrome DevTools Protocol domains declared by V8. Chrome DevTools Protocol domain provides an interface for interacting with one of the runtime agents used to inspect the application state and listen to the run-time events.

session.disconnect()

Added in: v8.0.0

Immediately close the session. All pending message callbacks will be called with an error. session.connect() will need to be called to be able to send messages again. Reconnected session will lose all inspector state, such as enabled agents or configured breakpoints.

Example usage

CPU Profiler

Apart from the debugger, various V8 Profilers are available through the DevTools protocol. Here's a simple example showing how to use the CPU profiler:

```
const inspector = require('inspector');
const fs = require('fs');
const session = new inspector.Session();
session.connect();

session.post('Profiler.enable', () => {
  session.post('Profiler.start', () => {
    // invoke business logic under measurement here...

    // some time later...
    session.post('Profiler.stop', (err, { profile }) => {
      // write profile to disk, upload, etc.
      if (!err) {
        fs.writeFileSync('./profile.cpuprofile', JSON.stringify(profile));
      }
    });
  });
});
```

Internationalization Support

Node.js has many features that make it easier to write internationalized programs. Some of them are:

- Locale-sensitive or Unicode-aware functions in the ECMAScript Language Specification:
  - String.prototype.normalize()
  - String.prototype.toLowerCase()
  - String.prototype.toUpperCase()
- All functionality described in the ECMAScript Internationalization API Specification (aka ECMA-402):
  - Intl object
  - Locale-sensitive methods like String.prototype.localeCompare() and Date.prototype.toLocaleString()
- The WHATWG URL parser's internationalized domain names (IDNs) support
- require('buffer').transcode()
- More accurate REPL line editing
- require('util').TextDecoder
- RegExp Unicode Property Escapes

Node.js (and its underlying V8 engine) uses ICU to implement these features in native C/C++ code. However, some of them require a very large ICU data file in order to support all locales of the world. Because it is expected that most Node.js users will make use of only a small portion of ICU functionality, only a subset of the full ICU data set is provided by Node.js by default. Several options are provided for customizing and expanding the ICU data set either when building or running Node.js.

Options for building Node.js

To control how ICU is used in Node.js, four `configure` options are available during compilation. Additional details on how to compile Node.js are documented in BUILDING.md.

- `--with-intl=none` / `--without-intl`
- `--with-intl=system-icu`
- `--with-intl=small-icu` (default)
- `--with-intl=full-icu`

An overview of available Node.js and JavaScript features for each `configure` option:

	`none`	`system-icu`	`small-icu`	`full-icu`
String.prototype.normalize()	none (function is no-op)	full	full	full
`String.prototype.to*Case()`	full	full	full	full
Intl	none (object does not exist)	partial/full (depends on OS)	partial (English-only)	full
String.prototype.localeCompare()	partial (not locale-aware)	full	full	full
`String.prototype.toLocale*Case()`	partial (not locale-aware)	full	full	full
Number.prototype.toLocaleString()	partial (not locale-aware)	partial/full (depends on OS)	partial (English-only)	full
`Date.prototype.toLocale*String()`	partial (not locale-aware)	partial/full (depends on OS)	partial (English-only)	full
WHATWG URL Parser	partial (no IDN support)	full	full	full
require('buffer').transcode()	none (function does not exist)	full	full	full
REPL	partial (inaccurate line editing)	full	full	full
require('util').TextDecoder	partial (basic encodings support)	partial/full (depends on OS)	partial (Unicode-only)	full
RegExp Unicode Property Escapes	none (invalid RegExp error)	full	full	full

The "(not locale-aware)" designation denotes that the function carries out its operation just like the non-`Locale` version of the function, if one exists. For example, under `none` mode, `Date.prototype.toLocaleString()`'s operation is identical to that of `Date.prototype.toString()`.

Disable all internationalization features (`none`)

If this option is chosen, most internationalization features mentioned above will be **unavailable** in the resulting `node` binary.

Build with a pre-installed ICU (`system-icu`)

Node.js can link against an ICU build already installed on the system. In fact, most Linux distributions already come with ICU installed, and this option would make it possible to reuse the same set of data used by other components in the OS.

Functionalities that only require the ICU library itself, such as String.prototype.normalize() and the WHATWG URL parser, are fully supported under `system-icu`. Features that require ICU locale data in addition, such as Intl.DateTimeFormat *may* be fully or partially supported, depending on the completeness of the ICU data installed on the system.

Embed a limited set of ICU data (`small-icu`)

This option makes the resulting binary link against the ICU library statically, and includes a subset of ICU data (typically only the English locale) within the `node` executable.

Functionalities that only require the ICU library itself, such as String.prototype.normalize() and the WHATWG URL parser, are fully supported under small-icu. Features that require ICU locale data in addition, such as Intl.DateTimeFormat, generally only work with the English locale:

```
const january = new Date(9e8);
const english = new Intl.DateTimeFormat('en', { month: 'long' });
const spanish = new Intl.DateTimeFormat('es', { month: 'long' });

console.log(english.format(january));
// Prints "January"
console.log(spanish.format(january));
// Prints "M01" on small-icu
// Should print "enero"
```

This mode provides a good balance between features and binary size, and it is the default behavior if no --with-intl flag is passed. The official binaries are also built in this mode.

Providing ICU data at runtime

If the small-icu option is used, one can still provide additional locale data at runtime so that the JS methods would work for all ICU locales. Assuming the data file is stored at /some/directory, it can be made available to ICU through either:

- The NODE_ICU_DATA environment variable:

```
env NODE_ICU_DATA=/some/directory node
```

- The --icu-data-dir CLI parameter:

```
node --icu-data-dir=/some/directory
```

(If both are specified, the --icu-data-dir CLI parameter takes precedence.)

ICU is able to automatically find and load a variety of data formats, but the data must be appropriate for the ICU version, and the file correctly named. The most common name for the data file is icudt5X[bl].dat, where 5X denotes the intended ICU version, and b or l indicates the system's endianness. Check "ICU Data" article in the ICU User Guide for other supported formats and more details on ICU data in general.

The full-icu npm module can greatly simplify ICU data installation by detecting the ICU version of the running node executable and downloading the appropriate data file. After installing the module through npm i full-icu, the data file will be available at ./node_modules/full-icu. This path can be then passed either to NODE_ICU_DATA or --icu-data-dir as shown above to enable full Intl support.

Embed the entire ICU (full-icu)

This option makes the resulting binary link against ICU statically and include a full set of ICU data. A binary created this way has no further external dependencies and supports all locales, but might be rather large. See BUILDING.md on how to compile a binary using this mode.

Detecting internationalization support

To verify that ICU is enabled at all (system-icu, small-icu, or full-icu), simply checking the existence of Intl should suffice:

```
const hasICU = typeof Intl === 'object';
```

Alternatively, checking for process.versions.icu, a property defined only when ICU is enabled, works too:

```
const hasICU = typeof process.versions.icu === 'string';
```

To check for support for a non-English locale (i.e. full-icu or system-icu), Intl.DateTimeFormat can be a good distinguishing factor:

```
const hasFullICU = (() => {
  try {
    const january = new Date(9e8);
    const spanish = new Intl.DateTimeFormat('es', { month: 'long' });
    return spanish.format(january) === 'enero';
  } catch (err) {
    return false;
  }
})();
```

For more verbose tests for `Intl` support, the following resources may be found to be helpful:

- btest402 : Generally used to check whether Node.js with `Intl` support is built correctly.
- Test262 : ECMAScript's official conformance test suite includes a section dedicated to ECMA-402.

Modules

Stability: 2 - Stable

In the Node.js module system, each file is treated as a separate module. For example, consider a file named `foo.js` :

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

On the first line, `foo.js` loads the module `circle.js` that is in the same directory as `foo.js` .

Here are the contents of `circle.js` :

```
const { PI } = Math;

exports.area = (r) => PI * r ** 2;

exports.circumference = (r) => 2 * PI * r;
```

The module `circle.js` has exported the functions `area()` and `circumference()` . Functions and objects are added to the root of a module by specifying additional properties on the special `exports` object.

Variables local to the module will be private, because the module is wrapped in a function by Node.js (see module wrapper). In this example, the variable `PI` is private to `circle.js` .

The `module.exports` property can be assigned a new value (such as a function or object).

Below, `bar.js` makes use of the `square` module, which exports a Square class:

```
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is ${mySquare.area()}`);
```

The `square` module is defined in `square.js` :

```
// assigning to exports will not modify module, must use module.exports
module.exports = class Square {
  constructor(width) {
    this.width = width;
  }

  area() {
    return this.width ** 2;
  }
};
```

The module system is implemented in the `require('module')` module.

Accessing the main module

When a file is run directly from Node.js, `require.main` is set to its `module`. That means that it is possible to determine whether a file has been run directly by testing `require.main === module`.

For a file `foo.js`, this will be `true` if run via `node foo.js`, but `false` if run by `require('./foo')`.

Because `module` provides a `filename` property (normally equivalent to `__filename`), the entry point of the current application can be obtained by checking `require.main.filename`.

Addenda: Package Manager Tips

The semantics of Node.js's `require()` function were designed to be general enough to support a number of reasonable directory structures. Package manager programs such as `dpkg`, `rpm`, and `npm` will hopefully find it possible to build native packages from Node.js modules without modification.

Below we give a suggested directory structure that could work:

Let's say that we wanted to have the folder at /usr/lib/node/<some-package>/<some-version> hold the contents of a specific version of a package.

Packages can depend on one another. In order to install package `foo`, it may be necessary to install a specific version of package `bar`. The `bar` package may itself have dependencies, and in some cases, these may even collide or form cyclic dependencies.

Since Node.js looks up the `realpath` of any modules it loads (that is, resolves symlinks), and then looks for their dependencies in the `node_modules` folders as described here, this situation is very simple to resolve with the following architecture:

- `/usr/lib/node/foo/1.2.3/` - Contents of the `foo` package, version 1.2.3.
- `/usr/lib/node/bar/4.3.2/` - Contents of the `bar` package that `foo` depends on.
- `/usr/lib/node/foo/1.2.3/node_modules/bar` - Symbolic link to `/usr/lib/node/bar/4.3.2/`.
- `/usr/lib/node/bar/4.3.2/node_modules/*` - Symbolic links to the packages that `bar` depends on.

Thus, even if a cycle is encountered, or if there are dependency conflicts, every module will be able to get a version of its dependency that it can use.

When the code in the `foo` package does `require('bar')`, it will get the version that is symlinked into `/usr/lib/node/foo/1.2.3/node_modules/bar`. Then, when the code in the `bar` package calls `require('quux')`, it'll get the version that is symlinked into `/usr/lib/node/bar/4.3.2/node_modules/quux`.

Furthermore, to make the module lookup process even more optimal, rather than putting packages directly in `/usr/lib/node`, we could put them in /usr/lib/node_modules/<name>/<version>. Then Node.js will not bother looking for missing dependencies in `/usr/node_modules` or `/node_modules`.

In order to make modules available to the Node.js REPL, it might be useful to also add the `/usr/lib/node_modules` folder to the `$NODE_PATH` environment variable. Since the module lookups using `node_modules` folders are all relative, and based on the real path of the files making the calls to `require()`, the packages themselves can be anywhere.

All Together...

To get the exact filename that will be loaded when `require()` is called, use the `require.resolve()` function.

Putting together all of the above, here is the high-level algorithm in pseudocode of what `require.resolve()` does:

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with '/'
   a. set Y to be the filesystem root
3. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
4. LOAD_NODE_MODULES(X, dirname(Y))
5. THROW "not found"

LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.json is a file, parse X.json to a JavaScript Object.  STOP
4. If X.node is a file, load X.node as binary addon.  STOP

LOAD_INDEX(X)
1. If X/index.js is a file, load X/index.js as JavaScript text.  STOP
2. If X/index.json is a file, parse X/index.json to a JavaScript object. STOP
3. If X/index.node is a file, load X/index.node as binary addon.  STOP

LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
   d. LOAD_INDEX(M)
2. LOAD_INDEX(X)

LOAD_NODE_MODULES(X, START)
1. let DIRS=NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)

NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let I = count of PARTS - 1
3. let DIRS = []
4. while I >= 0,
   a. if PARTS[I] = "node_modules" CONTINUE
   b. DIR = path join(PARTS[0 .. I] + "node_modules")
   c. DIRS = DIRS + DIR
   d. let I = I - 1
5. return DIRS
```

Caching

Modules are cached after the first time they are loaded. This means (among other things) that every call to `require('foo')` will get exactly the same object returned, if it would resolve to the same file.

Multiple calls to `require('foo')` may not cause the module code to be executed multiple times. This is an important feature. With it, "partially done" objects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles.

To have a module execute code multiple times, export a function, and call that function.

Module Caching Caveats

Modules are cached based on their resolved filename. Since modules may resolve to a different filename based on the location of the calling module (loading from `node_modules` folders), it is not a *guarantee* that `require('foo')` will always return the exact same object, if it would resolve to different files.

Additionally, on case-insensitive file systems or operating systems, different resolved filenames can point to the same file, but the cache will still treat them as different modules and will reload the file multiple times. For example, `require('./foo')` and `require('./FOO')` return two different objects, irrespective of whether or not `./foo` and `./FOO` are the same file.

Core Modules

Node.js has several modules compiled into the binary. These modules are described in greater detail elsewhere in this documentation.

The core modules are defined within Node.js's source and are located in the `lib/` folder.

Core modules are always preferentially loaded if their identifier is passed to `require()`. For instance, `require('http')` will always return the built in HTTP module, even if there is a file by that name.

Cycles

When there are circular `require()` calls, a module might not have finished executing when it is returned.

Consider this situation:

`a.js`:

```
console.log('a starting');
exports.done = false;
const b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

`b.js`:

```
console.log('b starting');
exports.done = false;
const a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

`main.js`:

```
console.log('main starting');
const a = require('./a.js');
const b = require('./b.js');
console.log('in main, a.done = %j, b.done = %j', a.done, b.done);
```

When `main.js` loads `a.js`, then `a.js` in turn loads `b.js`. At that point, `b.js` tries to load `a.js`. In order to prevent an infinite loop, an **unfinished copy** of the `a.js` exports object is returned to the `b.js` module. `b.js` then finishes loading, and its `exports` object is provided to the `a.js` module.

By the time `main.js` has loaded both modules, they're both finished. The output of this program would thus be:

```
$ node main.js
main starting
a starting
b starting
in b, a.done = false
b done
in a, b.done = true
a done
in main, a.done = true, b.done = true
```

Careful planning is required to allow cyclic module dependencies to work correctly within an application.

File Modules

If the exact filename is not found, then Node.js will attempt to load the required filename with the added extensions: `.js`, `.json`, and finally `.node`.

`.js` files are interpreted as JavaScript text files, and `.json` files are parsed as JSON text files. `.node` files are interpreted as compiled addon modules loaded with `dlopen`.

A required module prefixed with `'/'` is an absolute path to the file. For example, `require('/home/marco/foo.js')` will load the file at `/home/marco/foo.js`.

A required module prefixed with `'./'` is relative to the file calling `require()`. That is, `circle.js` must be in the same directory as `foo.js` for `require('./circle')` to find it.

Without a leading '/', './', or '../' to indicate a file, the module must either be a core module or is loaded from a `node_modules` folder.

If the given path does not exist, `require()` will throw an Error with its `code` property set to `'MODULE_NOT_FOUND'`.

Folders as Modules

It is convenient to organize programs and libraries into self-contained directories, and then provide a single entry point to that library. There are three ways in which a folder may be passed to `require()` as an argument.

The first is to create a `package.json` file in the root of the folder, which specifies a `main` module. An example package.json file might look like this:

```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

If this was in a folder at `./some-library`, then `require('./some-library')` would attempt to load `./some-library/lib/some-library.js`.

This is the extent of Node.js's awareness of package.json files.

If the file specified by the `'main'` entry of `package.json` is missing and can not be resolved, Node.js will report the entire module as missing with the default error:

```
Error: Cannot find module 'some-library'
```

If there is no package.json file present in the directory, then Node.js will attempt to load an `index.js` or `index.node` file out of that directory. For example, if there was no package.json file in the above example, then `require('./some-library')` would attempt to load:

- `./some-library/index.js`
- `./some-library/index.node`

Loading from `node_modules` Folders

If the module identifier passed to `require()` is not a core module, and does not begin with `'/'`, `'../'`, or `'./'`, then Node.js starts at the parent directory of the current module, and adds `/node_modules`, and attempts to load the module from that location. Node will not append `node_modules` to a path already ending in `node_modules`.

If it is not found there, then it moves to the parent directory, and so on, until the root of the file system is reached.

For example, if the file at `'/home/ry/projects/foo.js'` called `require('bar.js')`, then Node.js would look in the following locations, in this order:

- `/home/ry/projects/node_modules/bar.js`
- `/home/ry/node_modules/bar.js`
- `/home/node_modules/bar.js`
- `/node_modules/bar.js`

This allows programs to localize their dependencies, so that they do not clash.

It is possible to require specific files or sub modules distributed with a module by including a path suffix after the module name. For instance `require('example-module/path/to/file')` would resolve `path/to/file` relative to where `example-module` is located. The suffixed path follows the same module resolution semantics.

Loading from the global folders

If the `NODE_PATH` environment variable is set to a colon-delimited list of absolute paths, then Node.js will search those paths for modules if they are not found elsewhere.

On Windows, `NODE_PATH` is delimited by semicolons (`;`) instead of colons.

`NODE_PATH` was originally created to support loading modules from varying paths before the current module resolution algorithm was frozen.

`NODE_PATH` is still supported, but is less necessary now that the Node.js ecosystem has settled on a convention for locating dependent modules. Sometimes deployments that rely on `NODE_PATH` show surprising behavior when people are unaware that `NODE_PATH` must be set. Sometimes a module's dependencies change, causing a different version (or even a different module) to be loaded as the `NODE_PATH` is searched.

Additionally, Node.js will search in the following locations:

- 1: `$HOME/.node_modules`
- 2: `$HOME/.node_libraries`
- 3: `$PREFIX/lib/node`

Where `$HOME` is the user's home directory, and `$PREFIX` is Node.js's configured `node_prefix`.

These are mostly for historic reasons.

It is strongly encouraged to place dependencies in the local `node_modules` folder. These will be loaded faster, and more reliably.

The module wrapper

Before a module's code is executed, Node.js will wrap it with a function wrapper that looks like the following:

```
(function(exports, require, module, __filename, __dirname) {
// Module code actually lives in here
});
```

By doing this, Node.js achieves a few things:

- It keeps top-level variables (defined with `var`, `const` or `let`) scoped to the module rather than the global object.
- It helps to provide some global-looking variables that are actually specific to the module, such as:
  - The `module` and `exports` objects that the implementor can use to export values from the module.
  - The convenience variables `__filename` and `__dirname`, containing the module's absolute filename and directory path.

The module scope

__dirname

Added in: v0.1.27

- <string>

The directory name of the current module. This is the same as the path.dirname() of the __filename.

Example: running `node example.js` from `/Users/mjr`

```
console.log(__dirname);
// Prints: /Users/mjr
console.log(path.dirname(__filename));
// Prints: /Users/mjr
```

__filename

Added in: v0.0.1

- <string>

The file name of the current module. This is the resolved absolute path of the current module file.

For a main program this is not necessarily the same as the file name used in the command line.

See __dirname for the directory name of the current module.

Examples:

Running `node example.js` from `/Users/mjr`

```
console.log(__filename);
// Prints: /Users/mjr/example.js
console.log(__dirname);
// Prints: /Users/mjr
```

Given two modules: `a` and `b`, where `b` is a dependency of `a` and there is a directory structure of:

- `/Users/mjr/app/a.js`
- `/Users/mjr/app/node_modules/b/b.js`

References to `__filename` within `b.js` will return `/Users/mjr/app/node_modules/b/b.js` while references to `__filename` within `a.js` will return `/Users/mjr/app/a.js`.

exports

Added in: v0.1.12

A reference to the `module.exports` that is shorter to type. See the section about the exports shortcut for details on when to use `exports` and when to use `module.exports`.

module

Added in: v0.1.16

- <Object>

A reference to the current module, see the section about the module object . In particular, `module.exports` is used for defining what a module exports and makes available through `require()` .

require()

Added in: v0.1.13

- <Function>

To require modules.

require.cache

Added in: v0.3.0

- <Object>

Modules are cached in this object when they are required. By deleting a key value from this object, the next `require` will reload the module. Note that this does not apply to native addons , for which reloading will result in an Error.

require.extensions

Added in: v0.3.0 Deprecated since: v0.10.6

Stability: 0 - Deprecated

- <Object>

Instruct `require` on how to handle certain file extensions.

Process files with the extension `.sjs` as `.js` :

```
require.extensions['.sjs'] = require.extensions['.js'];
```

Deprecated In the past, this list has been used to load non-JavaScript modules into Node.js by compiling them on-demand. However, in practice, there are much better ways to do this, such as loading modules via some other Node.js program, or compiling them to JavaScript ahead of time.

Since the module system is locked, this feature will probably never go away. However, it may have subtle bugs and complexities that are best left untouched.

Note that the number of file system operations that the module system has to perform in order to resolve a `require(...)` statement to a filename scales linearly with the number of registered extensions.

In other words, adding extensions slows down the module loader and should be discouraged.

require.main

Added in: v0.1.17

- <Object>

The `Module` object representing the entry script loaded when the Node.js process launched. See "Accessing the main module" .

In `entry.js` script:

```
console.log(require.main);
```

```
node entry.js
```

```
Module {
  id: '.',
  exports: {},
  parent: null,
  filename: '/absolute/path/to/entry.js',
  loaded: false,
  children: [],
  paths:
   [ '/absolute/path/to/node_modules',
     '/absolute/path/node_modules',
     '/absolute/node_modules',
     '/node_modules' ] }
```

require.resolve(request[, options])

▸ History

- `request` <string> The module path to resolve.
- `options` <Object>
  - `paths` <Array> Paths to resolve module location from. If present, these paths are used instead of the default resolution paths. Note that each of these paths is used as a starting point for the module resolution algorithm, meaning that the `node_modules` hierarchy is checked from this location.
- Returns: <string>

Use the internal `require()` machinery to look up the location of a module, but rather than loading the module, just return the resolved filename.

require.resolve.paths(request)

Added in: v8.9.0

- `request` <string> The module path whose lookup paths are being retrieved.
- Returns: <Array> | <null>

Returns an array containing the paths searched during resolution of `request` or null if the `request` string references a core module, for example `http` or `fs`.

The `module` Object

Added in: v0.1.16

- <Object>

In each module, the `module` free variable is a reference to the object representing the current module. For convenience, `module.exports` is also accessible via the `exports` module-global. `module` is not actually a global but rather local to each module.

module.children

Added in: v0.1.16

- <Array>

The module objects required by this one.

module.exports

Added in: v0.1.16

- <Object>

The `module.exports` object is created by the Module system. Sometimes this is not acceptable; many want their module to be an instance of some class. To do this, assign the desired export object to `module.exports`. Note that assigning the desired object to `exports` will simply rebind the local `exports` variable, which is probably not what is desired.

For example suppose we were making a module called `a.js`

```
const EventEmitter = require('events');

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
setTimeout(() => {
  module.exports.emit('ready');
}, 1000);
```

Then in another file we could do

```
const a = require('./a');
a.on('ready', () => {
  console.log('module a is ready');
});
```

Note that assignment to `module.exports` must be done immediately. It cannot be done in any callbacks. This does not work:

x.js:

```
setTimeout(() => {
  module.exports = { a: 'hello' };
}, 0);
```

y.js:

```
const x = require('./x');
console.log(x.a);
```

exports shortcut

Added in: v0.1.16

The `exports` variable is available within a module's file-level scope, and is assigned the value of `module.exports` before the module is evaluated.

It allows a shortcut, so that `module.exports.f = ...` can be written more succinctly as `exports.f = ...`. However, be aware that like any variable, if a new value is assigned to `exports`, it is no longer bound to `module.exports`:

```
module.exports.hello = true; // Exported from require of module
exports = { hello: false };  // Not exported, only available in the module
```

When the `module.exports` property is being completely replaced by a new object, it is common to also reassign `exports`:

```
module.exports = exports = function Constructor() {
  // ... etc.
};
```

To illustrate the behavior, imagine this hypothetical implementation of `require()`, which is quite similar to what is actually done by `require()`:

```
function require(/* ... */) {
  const module = { exports: {} };
  ((module, exports) => {
    // Module code here. In this example, define a function.
    function someFunc() {}
    exports = someFunc;
    // At this point, exports is no longer a shortcut to module.exports, and
    // this module will still export an empty default object.
    module.exports = someFunc;
    // At this point, the module will now export someFunc, instead of the
    // default object.
  })(module, module.exports);
  return module.exports;
}
```

module.filename

Added in: v0.1.16

- <string>

The fully resolved filename to the module.

module.id

Added in: v0.1.16

- <string>

The identifier for the module. Typically this is the fully resolved filename.

module.loaded

Added in: v0.1.16

- <boolean>

Whether or not the module is done loading, or is in the process of loading.

module.parent

Added in: v0.1.16

- <Object> Module object

The module that first required this one.

module.paths

Added in: v0.4.0

- <string[]>

The search paths for the module.

module.require(id)

Added in: v0.5.1

- `id` <string>
- Returns: <Object> `module.exports` from the resolved module

The `module.require` method provides a way to load a module as if `require()` was called from the original module.

In order to do this, it is necessary to get a reference to the `module` object. Since `require()` returns the `module.exports`, and the `module` is typically *only* available within a specific module's code, it must be explicitly exported in order to be used.

The `Module` Object

Added in: v0.3.7

- <Object>

Provides general utility methods when interacting with instances of `Module` — the `module` variable often seen in file modules. Accessed via `require('module')`.

module.builtinModules

Added in: v9.3.0

- <string[]>

A list of the names of all modules provided by Node.js. Can be used to verify if a module is maintained by a third party or not.

Note that `module` in this context isn't the same object that's provided by the module wrapper. To access it, require the `Module` module:

```
const builtin = require('module').builtinModules;
```

Net

Stability: 2 - Stable

The `net` module provides an asynchronous network API for creating stream-based TCP or IPC servers (net.createServer()) and clients (net.createConnection()).

It can be accessed using:

```
const net = require('net');
```

IPC Support

The `net` module supports IPC with named pipes on Windows, and UNIX domain sockets on other operating systems.

Identifying paths for IPC connections

net.connect(), net.createConnection(), server.listen() and socket.connect() take a `path` parameter to identify IPC endpoints.

On UNIX, the local domain is also known as the UNIX domain. The path is a filesystem path name. It gets truncated to `sizeof(sockaddr_un.sun_path) - 1`, which varies on different operating system between 91 and 107 bytes. The typical values are 107 on Linux and 103 on macOS. The path is subject to the same naming conventions and permissions checks as would be done on file creation. It will be visible in the filesystem, and will *persist until unlinked*.

On Windows, the local domain is implemented using a named pipe. The path *must* refer to an entry in `\\?\pipe\` or `\\.\pipe\`. Any characters are permitted, but the latter may do some processing of pipe names, such as resolving `..` sequences. Despite appearances, the pipe name space is flat. Pipes will *not persist*, they are removed when the last reference to them is closed. Do not forget JavaScript string escaping requires paths to be specified with double-backslashes, such as:

```
net.createServer().listen(
  path.join('\\\\?\\pipe', process.cwd(), 'myctl'));
```

Class: net.Server

Added in: v0.1.90

This class is used to create a TCP or IPC server.

new net.Server([options][, connectionListener])

- Returns: <net.Server>

See net.createServer([options][, connectionListener]) .

`net.Server` is an EventEmitter with the following events:

Event: 'close'

Added in: v0.5.0

Emitted when the server closes. Note that if connections exist, this event is not emitted until all connections are ended.

Event: 'connection'

Added in: v0.1.90

- <net.Socket> The connection object

Emitted when a new connection is made. `socket` is an instance of `net.Socket`.

Event: 'error'

Added in: v0.1.90

- <Error>

Emitted when an error occurs. Unlike net.Socket , the 'close' event will **not** be emitted directly following this event unless server.close() is manually called. See the example in discussion of server.listen() .

Event: 'listening'

Added in: v0.1.90

Emitted when the server has been bound after calling server.listen() .

server.address()

Added in: v0.1.90

Returns the bound address, the address family name, and port of the server as reported by the operating system if listening on an IP socket. Useful to find which port was assigned when getting an OS-assigned address. Returns an object with `port`, `family`, and `address` properties: `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

For a server listening on a pipe or UNIX domain socket, the name is returned as a string.

Example:

```
const server = net.createServer((socket) => {
  socket.end('goodbye\n');
}).on('error', (err) => {
  // handle errors here
  throw err;
});

// grab an arbitrary unused port.
server.listen(() => {
  console.log('opened server on', server.address());
});
```

Don't call `server.address()` until the `'listening'` event has been emitted.

server.close([callback])

Added in: v0.1.90

- Returns: <net.Server>

Stops the server from accepting new connections and keeps existing connections. This function is asynchronous, the server is finally closed when all connections are ended and the server emits a `'close'` event. The optional `callback` will be called once the `'close'` event occurs. Unlike that event, it will be called with an Error as its only argument if the server was not open when it was closed.

Returns `server`.

server.connections

Added in: v0.2.0 Deprecated since: v0.9.7

> Stability: 0 - Deprecated: Use server.getConnections() instead.

The number of concurrent connections on the server.

This becomes `null` when sending a socket to a child with `child_process.fork()`. To poll forks and get current number of active connections use asynchronous `server.getConnections()` instead.

server.getConnections(callback)

Added in: v0.9.7

- Returns: <net.Server>

Asynchronously get the number of concurrent connections on the server. Works when sockets were sent to forks.

Callback should take two arguments `err` and `count`.

server.listen()

Start a server listening for connections. A `net.Server` can be a TCP or a IPC server depending on what it listens to.

Possible signatures:

- `server.listen(handle[, backlog][, callback])`
- `server.listen(options[, callback])`
- `server.listen(path[, backlog][, callback])` for IPC servers
- `server.listen([port][, host][, backlog][, callback])` for TCP servers

This function is asynchronous. When the server starts listening, the `'listening'` event will be emitted. The last parameter `callback` will be added as a listener for the `'listening'` event.

All listen() methods can take a backlog parameter to specify the maximum length of the queue of pending connections. The actual length will be determined by the OS through sysctl settings such as tcp_max_syn_backlog and somaxconn on Linux. The default value of this parameter is 511 (not 512).

All net.Socket are set to SO_REUSEADDR (See socket(7) for details).

The server.listen() method can be called again if and only if there was an error during the first server.listen() call or server.close() has been called. Otherwise, an ERR_SERVER_ALREADY_LISTEN error will be thrown.

One of the most common errors raised when listening is EADDRINUSE. This happens when another server is already listening on the requested port / path / handle. One way to handle this would be to retry after a certain amount of time:

```
server.on('error', (e) => {
  if (e.code === 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(() => {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

server.listen(handle[, backlog][, callback])

Added in: v0.5.10

- handle <Object>
- backlog <number> Common parameter of server.listen() functions
- callback <Function> Common parameter of server.listen() functions
- Returns: <net.Server>

Start a server listening for connections on a given handle that has already been bound to a port, a UNIX domain socket, or a Windows named pipe.

The handle object can be either a server, a socket (anything with an underlying _handle member), or an object with an fd member that is a valid file descriptor.

Listening on a file descriptor is not supported on Windows.

server.listen(options[, callback])

Added in: v0.11.14

- options <Object> Required. Supports the following properties:
  - port <number>
  - host <string>
  - path <string> Will be ignored if port is specified. See Identifying paths for IPC connections .
  - backlog <number> Common parameter of server.listen() functions.
  - exclusive <boolean> **Default:** false
- callback <Function> Common parameter of server.listen() functions.
- Returns: <net.Server>

If port is specified, it behaves the same as server.listen([port][, hostname][, backlog][, callback]) . Otherwise, if path is specified, it behaves the same as server.listen(path[, backlog][, callback]) . If none of them is specified, an error will be thrown.

If exclusive is false (default), then cluster workers will use the same underlying handle, allowing connection handling duties to be shared. When exclusive is true, the handle is not shared, and attempted port sharing results in an error. An example which listens on an exclusive port is shown below.

```
server.listen({
  host: 'localhost',
  port: 80,
  exclusive: true
});
```

server.listen(path[, backlog][, callback])

Added in: v0.1.90

- `path` <string> Path the server should listen to. See Identifying paths for IPC connections .
- `backlog` <number> Common parameter of `server.listen()` functions.
- `callback` <Function> Common parameter of `server.listen()` functions.
- Returns: <net.Server>

Start a IPC server listening for connections on the given `path`.

server.listen([port][, host][, backlog][, callback])

Added in: v0.1.90

- `port` <number>
- `host` <string>
- `backlog` <number> Common parameter of `server.listen()` functions.
- `callback` <Function> Common parameter of `server.listen()` functions.
- Returns: <net.Server>

Start a TCP server listening for connections on the given `port` and `host`.

If `port` is omitted or is 0, the operating system will assign an arbitrary unused port, which can be retrieved by using `server.address().port` after the `'listening'` event has been emitted.

If `host` is omitted, the server will accept connections on the unspecified IPv6 address (`::`) when IPv6 is available, or the unspecified IPv4 address (`0.0.0.0`) otherwise.

In most operating systems, listening to the unspecified IPv6 address (`::`) may cause the `net.Server` to also listen on the unspecified IPv4 address (`0.0.0.0`).

server.listening

Added in: v5.7.0

A Boolean indicating whether or not the server is listening for connections.

server.maxConnections

Added in: v0.2.0

Set this property to reject connections when the server's connection count gets high.

It is not recommended to use this option once a socket has been sent to a child with `child_process.fork()` .

server.ref()

Added in: v0.9.1

- Returns: <net.Server>

Opposite of `unref`, calling `ref` on a previously `unref`d server will *not* let the program exit if it's the only server left (the default behavior). If the server is `ref`d calling `ref` again will have no effect.

server.unref()

Added in: v0.9.1

- Returns: <net.Server>

Calling `unref` on a server will allow the program to exit if this is the only active server in the event system. If the server is already `unref`d calling `unref` again will have no effect.

Class: net.Socket

Added in: v0.3.4

This class is an abstraction of a TCP socket or a streaming IPC endpoint (uses named pipes on Windows, and UNIX domain sockets otherwise). A `net.Socket` is also a duplex stream, so it can be both readable and writable, and it is also a EventEmitter.

A `net.Socket` can be created by the user and used directly to interact with a server. For example, it is returned by net.createConnection(), so the user can use it to talk to the server.

It can also be created by Node.js and passed to the user when a connection is received. For example, it is passed to the listeners of a 'connection' event emitted on a net.Server, so the user can use it to interact with the client.

new net.Socket(options)

Added in: v0.3.4

Creates a new socket object.

- `options` <Object> Available options are:
  - `fd`: <number> If specified, wrap around an existing socket with the given file descriptor, otherwise a new socket will be created.
  - `allowHalfOpen` <boolean> Indicates whether half-opened TCP connections are allowed. See net.createServer() and the 'end' event for details. **Default:** `false`
  - `readable` <boolean> Allow reads on the socket when an `fd` is passed, otherwise ignored. **Default:** `false`
  - `writable` <boolean> Allow writes on the socket when an `fd` is passed, otherwise ignored. **Default:** `false`
- Returns: <net.Socket>

The newly created socket can be either a TCP socket or a streaming IPC endpoint, depending on what it connect() to.

Event: 'close'

Added in: v0.1.90

- `had_error` <boolean> `true` if the socket had a transmission error.

Emitted once the socket is fully closed. The argument `had_error` is a boolean which says if the socket was closed due to a transmission error.

Event: 'connect'

Added in: v0.1.90

Emitted when a socket connection is successfully established. See net.createConnection().

Event: 'data'

Added in: v0.1.90

- <Buffer>

Emitted when data is received. The argument `data` will be a `Buffer` or `String`. Encoding of data is set by socket.setEncoding().

Note that the **data will be lost** if there is no listener when a `Socket` emits a `'data'` event.

Event: 'drain'

Added in: v0.1.90

Emitted when the write buffer becomes empty. Can be used to throttle uploads.

See also: the return values of `socket.write()`

Event: 'end'

Added in: v0.1.90

Emitted when the other end of the socket sends a FIN packet, thus ending the readable side of the socket.

By default (`allowHalfOpen` is `false`) the socket will send a FIN packet back and destroy its file descriptor once it has written out its pending write queue. However, if `allowHalfOpen` is set to `true`, the socket will not automatically end() its writable side, allowing the user to write arbitrary amounts of data. The user must call end() explicitly to close the connection (i.e. sending a FIN packet back).

Event: 'error'

Added in: v0.1.90

- <Error>

Emitted when an error occurs. The `'close'` event will be called directly following this event.

Event: 'lookup'

▸ History

Emitted after resolving the hostname but before connecting. Not applicable to UNIX sockets.

- `err` <Error> | <null> The error object. See dns.lookup() .
- `address` <string> The IP address.
- `family` <string> | <null> The address type. See dns.lookup() .
- `host` <string> The hostname.

Event: 'timeout'

Added in: v0.1.90

Emitted if the socket times out from inactivity. This is only to notify that the socket has been idle. The user must manually close the connection.

See also: socket.setTimeout()

socket.address()

Added in: v0.1.90

Returns the bound address, the address family name and port of the socket as reported by the operating system. Returns an object with three properties, e.g. `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

socket.bufferSize

Added in: v0.3.8

`net.Socket` has the property that `socket.write()` always works. This is to help users get up and running quickly. The computer cannot always keep up with the amount of data that is written to a socket - the network connection simply might be too slow. Node.js will internally queue up the data written to a socket and send it out over the wire when it is possible. (Internally it is polling on the socket's file descriptor for being writable).

The consequence of this internal buffering is that memory may grow. This property shows the number of characters currently buffered to be written. (Number of characters is approximately equal to the number of bytes to be written, but the buffer may contain strings, and the strings are lazily encoded, so the exact number of bytes is not known.)

Users who experience large or growing `bufferSize` should attempt to "throttle" the data flows in their program with socket.pause() and

socket.resume() .

socket.bytesRead

Added in: v0.5.3

The amount of received bytes.

socket.bytesWritten

Added in: v0.5.3

The amount of bytes sent.

socket.connect()

Initiate a connection on a given socket.

Possible signatures:

- socket.connect(options[, connectListener])
- socket.connect(path[, connectListener]) for IPC connections.
- socket.connect(port[, host][, connectListener]) for TCP connections.
- Returns: <net.Socket> The socket itself.

This function is asynchronous. When the connection is established, the 'connect' event will be emitted. If there is a problem connecting, instead of a 'connect' event, an 'error' event will be emitted with the error passed to the 'error' listener. The last parameter `connectListener`, if supplied, will be added as a listener for the 'connect' event **once**.

socket.connect(options[, connectListener])

▸ History

- `options` <Object>
- `connectListener` <Function> Common parameter of socket.connect() methods. Will be added as a listener for the 'connect' event once.
- Returns: <net.Socket> The socket itself.

Initiate a connection on a given socket. Normally this method is not needed, the socket should be created and opened with net.createConnection() . Use this only when implementing a custom Socket.

For TCP connections, available `options` are:

- `port` <number> Required. Port the socket should connect to.
- `host` <string> Host the socket should connect to. **Default:** `'localhost'`
- `localAddress` <string> Local address the socket should connect from.
- `localPort` <number> Local port the socket should connect from.
- `family` <number> : Version of IP stack, can be either 4 or 6. **Default:** `4`
- `hints` <number> Optional dns.lookup() hints .
- `lookup` <Function> Custom lookup function. **Default:** dns.lookup()

For IPC connections, available `options` are:

- `path` <string> Required. Path the client should connect to. See Identifying paths for IPC connections . If provided, the TCP-specific options above are ignored.

Returns `socket`.

socket.connect(path[, connectListener])

- `path` <string> Path the client should connect to. See Identifying paths for IPC connections .
- `connectListener` <Function> Common parameter of socket.connect() methods. Will be added as a listener for the 'connect' event once.

- Returns: <net.Socket> The socket itself.

Initiate an IPC connection on the given socket.

Alias to socket.connect(options[, connectListener]) called with `{ path: path }` as `options`.

Returns `socket`.

socket.connect(port[, host][, connectListener])

Added in: v0.1.90

- `port` <number> Port the client should connect to.
- `host` <string> Host the client should connect to.
- `connectListener` <Function> Common parameter of socket.connect() methods. Will be added as a listener for the 'connect' event once.
- Returns: <net.Socket> The socket itself.

Initiate a TCP connection on the given socket.

Alias to socket.connect(options[, connectListener]) called with `{port: port, host: host}` as `options`.

Returns `socket`.

socket.connecting

Added in: v6.1.0

If `true` - socket.connect(options[, connectListener]) was called and haven't yet finished. Will be set to `false` before emitting `connect` event and/or calling socket.connect(options[, connectListener]) 's callback.

socket.destroy([exception])

Added in: v0.1.90

- Returns: <net.Socket>

Ensures that no more I/O activity happens on this socket. Only necessary in case of errors (parse error or so).

If `exception` is specified, an 'error' event will be emitted and any listeners for that event will receive `exception` as an argument.

socket.destroyed

A Boolean value that indicates if the connection is destroyed or not. Once a connection is destroyed no further data can be transferred using it.

socket.end([data][, encoding])

Added in: v0.1.90

- Returns: <net.Socket> The socket itself.

Half-closes the socket. i.e., it sends a FIN packet. It is possible the server will still send some data.

If `data` is specified, it is equivalent to calling `socket.write(data, encoding)` followed by socket.end() .

socket.localAddress

Added in: v0.9.6

The string representation of the local IP address the remote client is connecting on. For example, in a server listening on `'0.0.0.0'`, if a client connects on `'192.168.1.1'`, the value of `socket.localAddress` would be `'192.168.1.1'`.

socket.localPort

Added in: v0.9.6

The numeric representation of the local port. For example, `80` or `21`.

socket.pause()

- Returns: <net.Socket> The socket itself.

Pauses the reading of data. That is, `'data'` events will not be emitted. Useful to throttle back an upload.

socket.ref()

Added in: v0.9.1

- Returns: <net.Socket> The socket itself.

Opposite of `unref`, calling `ref` on a previously `unref`d socket will *not* let the program exit if it's the only socket left (the default behavior). If the socket is `ref`d calling `ref` again will have no effect.

socket.remoteAddress

Added in: v0.5.10

The string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`. Value may be `undefined` if the socket is destroyed (for example, if the client disconnected).

socket.remoteFamily

Added in: v0.11.14

The string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

socket.remotePort

Added in: v0.5.10

The numeric representation of the remote port. For example, `80` or `21`.

socket.resume()

- Returns: <net.Socket> The socket itself.

Resumes reading after a call to `socket.pause()`.

socket.setEncoding([encoding])

Added in: v0.1.90

- Returns: <net.Socket> The socket itself.

Set the encoding for the socket as a Readable Stream. See `readable.setEncoding()` for more information.

socket.setKeepAlive([enable][, initialDelay])

Added in: v0.1.92

- Returns: <net.Socket> The socket itself.

Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket. `enable` defaults to `false`.

Set `initialDelay` (in milliseconds) to set the delay between the last data packet received and the first keepalive probe. Setting 0 for initialDelay will leave the value unchanged from the default (or previous) setting. Defaults to `0`.

socket.setNoDelay([noDelay])

Added in: v0.1.90

- Returns: <net.Socket> The socket itself.

Disables the Nagle algorithm. By default TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting `true` for `noDelay` will immediately fire off data each time `socket.write()` is called. `noDelay` defaults to `true`.

socket.setTimeout(timeout[, callback])

Added in: v0.1.90

- Returns: <net.Socket> The socket itself.

Sets the socket to timeout after `timeout` milliseconds of inactivity on the socket. By default `net.Socket` do not have a timeout.

When an idle timeout is triggered the socket will receive a `'timeout'` event but the connection will not be severed. The user must manually call `socket.end()` or `socket.destroy()` to end the connection.

```
socket.setTimeout(3000);
socket.on('timeout', () => {
  console.log('socket timeout');
  socket.end();
});
```

If `timeout` is 0, then the existing idle timeout is disabled.

The optional `callback` parameter will be added as a one-time listener for the `'timeout'` event.

socket.unref()

Added in: v0.9.1

- Returns: <net.Socket> The socket itself.

Calling `unref` on a socket will allow the program to exit if this is the only active socket in the event system. If the socket is already `unref` d calling `unref` again will have no effect.

socket.write(data[, encoding][, callback])

Added in: v0.1.90

Sends data on the socket. The second parameter specifies the encoding in the case of a string — it defaults to UTF8 encoding.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is again free.

The optional `callback` parameter will be executed when the data is finally written out - this may not be immediately.

net.connect()

Aliases to `net.createConnection()`.

Possible signatures:

- `net.connect(options[, connectListener])`
- `net.connect(path[, connectListener])` for IPC connections.
- `net.connect(port[, host][, connectListener])` for TCP connections.

net.connect(options[, connectListener])

Added in: v0.7.0

Alias to `net.createConnection(options[, connectListener])`.

net.connect(path[, connectListener])

Added in: v0.1.90

Alias to net.createConnection(path[, connectListener]) .

net.connect(port[, host][, connectListener])

Added in: v0.1.90

Alias to net.createConnection(port[, host][, connectListener]) .

net.createConnection()

A factory function, which creates a new net.Socket , immediately initiates connection with socket.connect() , then returns the net.Socket that starts the connection.

When the connection is established, a 'connect' event will be emitted on the returned socket. The last parameter connectListener , if supplied, will be added as a listener for the 'connect' event **once**.

Possible signatures:

- net.createConnection(options[, connectListener])
- net.createConnection(path[, connectListener]) for IPC connections.
- net.createConnection(port[, host][, connectListener]) for TCP connections.

The net.connect() function is an alias to this function.

net.createConnection(options[, connectListener])

Added in: v0.1.90

- options <Object> Required. Will be passed to both the new net.Socket([options]) call and the socket.connect(options[, connectListener]) method.
- connectListener <Function> Common parameter of the net.createConnection() functions. If supplied, will be added as a listener for the 'connect' event on the returned socket once.
- Returns: <net.Socket> The newly created socket used to start the connection.

For available options, see new net.Socket([options]) and socket.connect(options[, connectListener]) .

Additional options:

- timeout <number> If set, will be used to call socket.setTimeout(timeout) after the socket is created, but before it starts the connection.

Following is an example of a client of the echo server described in the net.createServer() section:

```
const net = require('net');
const client = net.createConnection({ port: 8124 }, () => {
  //'connect' listener
  console.log('connected to server!');
  client.write('world!\r\n');
});
client.on('data', (data) => {
  console.log(data.toString());
  client.end();
});
client.on('end', () => {
  console.log('disconnected from server');
});
```

To connect on the socket /tmp/echo.sock the second line would just be changed to

```
const client = net.createConnection({ path: '/tmp/echo.sock' });
```

net.createConnection(path[, connectListener])

Added in: v0.1.90

- `path` <string> Path the socket should connect to. Will be passed to socket.connect(path[, connectListener]) . See Identifying paths for IPC connections .
- `connectListener` <Function> Common parameter of the net.createConnection() functions, an "once" listener for the `'connect'` event on the initiating socket. Will be passed to socket.connect(path[, connectListener]) .
- Returns: <net.Socket> The newly created socket used to start the connection.

Initiates an IPC connection.

This function creates a new net.Socket with all options set to default, immediately initiates connection with socket.connect(path[, connectListener]) , then returns the `net.Socket` that starts the connection.

net.createConnection(port[, host][, connectListener])

Added in: v0.1.90

- `port` <number> Port the socket should connect to. Will be passed to socket.connect(port[, host][, connectListener]) .
- `host` <string> Host the socket should connect to. Will be passed to socket.connect(port[, host][, connectListener]) . **Default:** `'localhost'`
- `connectListener` <Function> Common parameter of the net.createConnection() functions, an "once" listener for the `'connect'` event on the initiating socket. Will be passed to socket.connect(path[, connectListener]) .
- Returns: <net.Socket> The newly created socket used to start the connection.

Initiates a TCP connection.

This function creates a new net.Socket with all options set to default, immediately initiates connection with socket.connect(port[, host][, connectListener]) , then returns the `net.Socket` that starts the connection.

net.createServer([options][, connectionListener])

Added in: v0.5.0

Creates a new TCP or IPC server.

- `options` <Object>
  - `allowHalfOpen` <boolean> Indicates whether half-opened TCP connections are allowed. **Default:** `false`
  - `pauseOnConnect` <boolean> Indicates whether the socket should be paused on incoming connections. **Default:** `false`
- `connectionListener` <Function> Automatically set as a listener for the 'connection' event.
- Returns: <net.Server>

If `allowHalfOpen` is set to `true`, when the other end of the socket sends a FIN packet, the server will only send a FIN packet back when socket.end() is explicitly called, until then the connection is half-closed (non-readable but still writable). See 'end' event and RFC 1122 (section 4.2.2.13) for more information.

If `pauseOnConnect` is set to `true`, then the socket associated with each incoming connection will be paused, and no data will be read from its handle. This allows connections to be passed between processes without any data being read by the original process. To begin reading data from a paused socket, call socket.resume() .

The server can be a TCP server or a IPC server, depending on what it listen() to.

Here is an example of an TCP echo server which listens for connections on port 8124:

```
const net = require('net');
const server = net.createServer((c) => {
  // 'connection' listener
  console.log('client connected');
  c.on('end', () => {
    console.log('client disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});
server.on('error', (err) => {
  throw err;
});
server.listen(8124, () => {
  console.log('server bound');
});
```

Test this by using `telnet`:

```
$ telnet localhost 8124
```

To listen on the socket `/tmp/echo.sock` the third line from the last would just be changed to

```
server.listen('/tmp/echo.sock', () => {
  console.log('server bound');
});
```

Use `nc` to connect to a UNIX domain socket server:

```
$ nc -U /tmp/echo.sock
```

net.isIP(input)

Added in: v0.3.0

Tests if input is an IP address. Returns 0 for invalid strings, returns 4 for IP version 4 addresses, and returns 6 for IP version 6 addresses.

net.isIPv4(input)

Added in: v0.3.0

Returns true if input is a version 4 IP address, otherwise returns false.

net.isIPv6(input)

Added in: v0.3.0

Returns true if input is a version 6 IP address, otherwise returns false.

OS

Stability: 2 - Stable

The `os` module provides a number of operating system-related utility methods. It can be accessed using:

```
const os = require('os');
```

os.EOL

Added in: v0.7.8

- <string>

A string constant defining the operating system-specific end-of-line marker:

- `\n` on POSIX
- `\r\n` on Windows

os.arch()

Added in: v0.5.0

- Returns: <string>

The `os.arch()` method returns a string identifying the operating system CPU architecture for which the Node.js binary was compiled.

The current possible values are: `'arm'`, `'arm64'`, `'ia32'`, `'mips'`, `'mipsel'`, `'ppc'`, `'ppc64'`, `'s390'`, `'s390x'`, `'x32'`, and `'x64'`.

Equivalent to `process.arch`.

os.constants

Added in: v6.3.0

- <Object>

Returns an object containing commonly used operating system specific constants for error codes, process signals, and so on. The specific constants currently defined are described in OS Constants.

os.cpus()

Added in: v0.3.3

- Returns: <Array>

The `os.cpus()` method returns an array of objects containing information about each logical CPU core.

The properties included on each object include:

- `model` <string>
- `speed` <number> (in MHz)
- `times` <Object>
  - `user` <number> The number of milliseconds the CPU has spent in user mode.
  - `nice` <number> The number of milliseconds the CPU has spent in nice mode.
  - `sys` <number> The number of milliseconds the CPU has spent in sys mode.
  - `idle` <number> The number of milliseconds the CPU has spent in idle mode.
  - `irq` <number> The number of milliseconds the CPU has spent in irq mode.

```
[
  {
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 252020,
      nice: 0,
      sys: 30340,
```

```
    idle: 1070356870,
    irq: 0
  }
},
{
  model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times: {
    user: 306960,
    nice: 0,
    sys: 26980,
    idle: 1071569080,
    irq: 0
  }
},
{
  model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times: {
    user: 248450,
    nice: 0,
    sys: 21750,
    idle: 1070919370,
    irq: 0
  }
},
{
  model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times: {
    user: 256880,
    nice: 0,
    sys: 19430,
    idle: 1070905480,
    irq: 20
  }
},
{
  model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times: {
    user: 511580,
    nice: 20,
    sys: 40900,
    idle: 1070842510,
    irq: 0
  }
},
{
  model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times: {
    user: 291660,
    nice: 0,
    sys: 34360,
    idle: 1070888000,
    irq: 10
  }
```

```
  },
  {
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 308260,
      nice: 0,
      sys: 55410,
      idle: 1071129970,
      irq: 880
    }
  },
  {
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 266450,
      nice: 1480,
      sys: 34920,
      idle: 1072572010,
      irq: 30
    }
  }
]
```

Because `nice` values are UNIX-specific, on Windows the `nice` values of all processors are always 0.

os.endianness()

Added in: v0.9.4

- Returns: <string>

The `os.endianness()` method returns a string identifying the endianness of the CPU*for which the Node.js binary was compiled.*

Possible values are:

- `'BE'` for big endian
- `'LE'` for little endian.

os.freemem()

Added in: v0.3.3

- Returns: <integer>

The `os.freemem()` method returns the amount of free system memory in bytes as an integer.

os.homedir()

Added in: v2.3.0

- Returns: <string>

The `os.homedir()` method returns the home directory of the current user as a string.

os.hostname()

Added in: v0.3.3

- Returns: <string>

The `os.hostname()` method returns the hostname of the operating system as a string.

os.loadavg()

Added in: v0.3.3

- Returns: <Array>

The `os.loadavg()` method returns an array containing the 1, 5, and 15 minute load averages.

The load average is a measure of system activity, calculated by the operating system and expressed as a fractional number. As a rule of thumb, the load average should ideally be less than the number of logical CPUs in the system.

The load average is a UNIX-specific concept with no real equivalent on Windows platforms. On Windows, the return value is always `[0, 0, 0]`.

os.networkInterfaces()

Added in: v0.6.0

- Returns: <Object>

The `os.networkInterfaces()` method returns an object containing only network interfaces that have been assigned a network address.

Each key on the returned object identifies a network interface. The associated value is an array of objects that each describe an assigned network address.

The properties available on the assigned network address object include:

- `address` <string> The assigned IPv4 or IPv6 address
- `netmask` <string> The IPv4 or IPv6 network mask
- `family` <string> Either `IPv4` or `IPv6`
- `mac` <string> The MAC address of the network interface
- `internal` <boolean> `true` if the network interface is a loopback or similar interface that is not remotely accessible; otherwise `false`
- `scopeid` <number> The numeric IPv6 scope ID (only specified when `family` is `IPv6`)
- `cidr` <string> The assigned IPv4 or IPv6 address with the routing prefix in CIDR notation. If the `netmask` is invalid, this property is set to `null`

```
{
  lo: [
    {
      address: '127.0.0.1',
      netmask: '255.0.0.0',
      family: 'IPv4',
      mac: '00:00:00:00:00:00',
      internal: true,
      cidr: '127.0.0.1/8'
    },
    {
      address: '::1',
      netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
      family: 'IPv6',
      mac: '00:00:00:00:00:00',
      internal: true,
      cidr: '::1/128'
    }
  ],
  eth0: [
    {
      address: '192.168.1.108',
      netmask: '255.255.255.0',
      family: 'IPv4',
      mac: '01:02:03:0a:0b:0c',
      internal: false,
      cidr: '192.168.1.108/24'
    },
    {
      address: 'fe80::a00:27ff:fe4e:66a1',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: '01:02:03:0a:0b:0c',
      internal: false,
      cidr: 'fe80::a00:27ff:fe4e:66a1/64'
    }
  ]
}
```

os.platform()

Added in: v0.5.0

- Returns: <string>

The `os.platform()` method returns a string identifying the operating system platform as set during compile time of Node.js.

Currently possible values are:

- `'aix'`
- `'darwin'`
- `'freebsd'`
- `'linux'`
- `'openbsd'`
- `'sunos'`
- `'win32'`

Equivalent to process.platform.

The value `'android'` may also be returned if the Node.js is built on the Android operating system. However, Android support in Node.js is considered to be experimental at this time.

os.release()

Added in: v0.3.3

- Returns: <string>

The `os.release()` method returns a string identifying the operating system release.

On POSIX systems, the operating system release is determined by calling uname(3). On Windows, `GetVersionExW()` is used. Please see https://en.wikipedia.org/wiki/Uname#Examples for more information.

os.tmpdir()

▸ History

- Returns: <string>

The `os.tmpdir()` method returns a string specifying the operating system's default directory for temporary files.

os.totalmem()

Added in: v0.3.3

- Returns: <integer>

The `os.totalmem()` method returns the total amount of system memory in bytes as an integer.

os.type()

Added in: v0.3.3

- Returns: <string>

The `os.type()` method returns a string identifying the operating system name as returned by uname(3). For example `'Linux'` on Linux, `'Darwin'` on macOS and `'Windows_NT'` on Windows.

Please see https://en.wikipedia.org/wiki/Uname#Examples for additional information about the output of running uname(3) on various operating systems.

os.uptime()

Added in: v0.3.3

- Returns: <integer>

The `os.uptime()` method returns the system uptime in number of seconds.

On Windows the returned value includes fractions of a second. Use `Math.floor()` to get whole seconds.

os.userInfo(options)

Added in: v6.0.0

- `options` <Object>
  - `encoding` <string> Character encoding used to interpret resulting strings. If `encoding` is set to `'buffer'`, the `username`, `shell`, and `homedir` values will be `Buffer` instances. **Default:** `'utf8'`
- Returns: <Object>

The `os.userInfo()` method returns information about the currently effective user — on POSIX platforms, this is typically a subset of the password file. The returned object includes the `username`, `uid`, `gid`, `shell`, and `homedir`. On Windows, the `uid` and `gid` fields are `-1`, and `shell` is

null .

The value of homedir returned by os.userInfo() is provided by the operating system. This differs from the result of os.homedir() , which queries several environment variables for the home directory before falling back to the operating system response.

OS Constants

The following constants are exported by os.constants .

Not all constants will be available on every operating system.

Signal Constants

▸ History

The following signal constants are exported by os.constants.signals :

Constant	Description
SIGHUP	Sent to indicate when a controlling terminal is closed or a parent process exits.
SIGINT	Sent to indicate when a user wishes to interrupt a process ((Ctrl+C)).
SIGQUIT	Sent to indicate when a user wishes to terminate a process and perform a core dump.
SIGILL	Sent to a process to notify that it has attempted to perform an illegal, malformed, unknown, or privileged instruction.
SIGTRAP	Sent to a process when an exception has occurred.
SIGABRT	Sent to a process to request that it abort.
SIGIOT	Synonym for SIGABRT
SIGBUS	Sent to a process to notify that it has caused a bus error.
SIGFPE	Sent to a process to notify that it has performed an illegal arithmetic operation.
SIGKILL	Sent to a process to terminate it immediately.
SIGUSR1 SIGUSR2	Sent to a process to identify user-defined conditions.
SIGSEGV	Sent to a process to notify of a segmentation fault.
SIGPIPE	Sent to a process when it has attempted to write to a disconnected pipe.
SIGALRM	Sent to a process when a system timer elapses.
SIGTERM	Sent to a process to request termination.
SIGCHLD	Sent to a process when a child process terminates.
SIGSTKFLT	Sent to a process to indicate a stack fault on a coprocessor.
SIGCONT	Sent to instruct the operating system to continue a paused process.
SIGSTOP	Sent to instruct the operating system to halt a process.
	Sent to a process to request it to stop.

SIGTSTP	
SIGBREAK	Sent to indicate when a user wishes to interrupt a process.
SIGTTIN	Sent to a process when it reads from the TTY while in the background.
SIGTTOU	Sent to a process when it writes to the TTY while in the background.
SIGURG	Sent to a process when a socket has urgent data to read.
SIGXCPU	Sent to a process when it has exceeded its limit on CPU usage.
SIGXFSZ	Sent to a process when it grows a file larger than the maximum allowed.
SIGVTALRM	Sent to a process when a virtual timer has elapsed.
SIGPROF	Sent to a process when a system timer has elapsed.
SIGWINCH	Sent to a process when the controlling terminal has changed its size.
SIGIO	Sent to a process when I/O is available.
SIGPOLL	Synonym for `SIGIO`
SIGLOST	Sent to a process when a file lock has been lost.
SIGPWR	Sent to a process to notify of a power failure.
SIGINFO	Synonym for `SIGPWR`
SIGSYS	Sent to a process to notify of a bad argument.
SIGUNUSED	Synonym for `SIGSYS`

Error Constants

The following error constants are exported by `os.constants.errno`:

POSIX Error Constants

Constant	Description
E2BIG	Indicates that the list of arguments is longer than expected.
EACCES	Indicates that the operation did not have sufficient permissions.
EADDRINUSE	Indicates that the network address is already in use.
EADDRNOTAVAIL	Indicates that the network address is currently unavailable for use.
EAFNOSUPPORT	Indicates that the network address family is not supported.
EAGAIN	Indicates that there is currently no data available and to try the operation again later.
EALREADY	Indicates that the socket already has a pending connection in progress.
EBADF	Indicates that a file descriptor is not valid.
EBADMSG	Indicates an invalid data message.

EBUSY	Indicates that a device or resource is busy.
ECANCELED	Indicates that an operation was canceled.
ECHILD	Indicates that there are no child processes.
ECONNABORTED	Indicates that the network connection has been aborted.
ECONNREFUSED	Indicates that the network connection has been refused.
ECONNRESET	Indicates that the network connection has been reset.
EDEADLK	Indicates that a resource deadlock has been avoided.
EDESTADDRREQ	Indicates that a destination address is required.
EDOM	Indicates that an argument is out of the domain of the function.
EDQUOT	Indicates that the disk quota has been exceeded.
EEXIST	Indicates that the file already exists.
EFAULT	Indicates an invalid pointer address.
EFBIG	Indicates that the file is too large.
EHOSTUNREACH	Indicates that the host is unreachable.
EIDRM	Indicates that the identifier has been removed.
EILSEQ	Indicates an illegal byte sequence.
EINPROGRESS	Indicates that an operation is already in progress.
EINTR	Indicates that a function call was interrupted.
EINVAL	Indicates that an invalid argument was provided.
EIO	Indicates an otherwise unspecified I/O error.
EISCONN	Indicates that the socket is connected.
EISDIR	Indicates that the path is a directory.
ELOOP	Indicates too many levels of symbolic links in a path.
EMFILE	Indicates that there are too many open files.
EMLINK	Indicates that there are too many hard links to a file.
EMSGSIZE	Indicates that the provided message is too long.
EMULTIHOP	Indicates that a multihop was attempted.
ENAMETOOLONG	Indicates that the filename is too long.
ENETDOWN	Indicates that the network is down.

ENETRESET	Indicates that the connection has been aborted by the network.
ENETUNREACH	Indicates that the network is unreachable.
ENFILE	Indicates too many open files in the system.
ENOBUFS	Indicates that no buffer space is available.
ENODATA	Indicates that no message is available on the stream head read queue.
ENODEV	Indicates that there is no such device.
ENOENT	Indicates that there is no such file or directory.
ENOEXEC	Indicates an exec format error.
ENOLCK	Indicates that there are no locks available.
ENOLINK	Indications that a link has been severed.
ENOMEM	Indicates that there is not enough space.
ENOMSG	Indicates that there is no message of the desired type.
ENOPROTOOPT	Indicates that a given protocol is not available.
ENOSPC	Indicates that there is no space available on the device.
ENOSR	Indicates that there are no stream resources available.
ENOSTR	Indicates that a given resource is not a stream.
ENOSYS	Indicates that a function has not been implemented.
ENOTCONN	Indicates that the socket is not connected.
ENOTDIR	Indicates that the path is not a directory.
ENOTEMPTY	Indicates that the directory is not empty.
ENOTSOCK	Indicates that the given item is not a socket.
ENOTSUP	Indicates that a given operation is not supported.
ENOTTY	Indicates an inappropriate I/O control operation.
ENXIO	Indicates no such device or address.
EOPNOTSUPP	Indicates that an operation is not supported on the socket. Note that while ENOTSUP and EOPNOTSUPP have the same value on Linux, according to POSIX.1 these error values should be distinct.)
EOVERFLOW	Indicates that a value is too large to be stored in a given data type.
EPERM	Indicates that the operation is not permitted.
EPIPE	Indicates a broken pipe.
EPROTO	Indicates a protocol error.

`EPROTONOSUPPORT`	Indicates that a protocol is not supported.
`EPROTOTYPE`	Indicates the wrong type of protocol for a socket.
`ERANGE`	Indicates that the results are too large.
`EROFS`	Indicates that the file system is read only.
`ESPIPE`	Indicates an invalid seek operation.
`ESRCH`	Indicates that there is no such process.
`ESTALE`	Indicates that the file handle is stale.
`ETIME`	Indicates an expired timer.
`ETIMEDOUT`	Indicates that the connection timed out.
`ETXTBSY`	Indicates that a text file is busy.
`EWOULDBLOCK`	Indicates that the operation would block.
`EXDEV`	Indicates an improper link.

Windows Specific Error Constants

The following error codes are specific to the Windows operating system:

Constant	Description
`WSAEINTR`	Indicates an interrupted function call.
`WSAEBADF`	Indicates an invalid file handle.
`WSAEACCES`	Indicates insufficient permissions to complete the operation.
`WSAEFAULT`	Indicates an invalid pointer address.
`WSAEINVAL`	Indicates that an invalid argument was passed.
`WSAEMFILE`	Indicates that there are too many open files.
`WSAEWOULDBLOCK`	Indicates that a resource is temporarily unavailable.
`WSAEINPROGRESS`	Indicates that an operation is currently in progress.
`WSAEALREADY`	Indicates that an operation is already in progress.
`WSAENOTSOCK`	Indicates that the resource is not a socket.
`WSAEDESTADDRREQ`	Indicates that a destination address is required.
`WSAEMSGSIZE`	Indicates that the message size is too long.
`WSAEPROTOTYPE`	Indicates the wrong protocol type for the socket.
`WSAENOPROTOOPT`	Indicates a bad protocol option.

WSAEPROTONOSUPPORT	Indicates that the protocol is not supported.
WSAESOCKTNOSUPPORT	Indicates that the socket type is not supported.
WSAEOPNOTSUPP	Indicates that the operation is not supported.
WSAEPFNOSUPPORT	Indicates that the protocol family is not supported.
WSAEAFNOSUPPORT	Indicates that the address family is not supported.
WSAEADDRINUSE	Indicates that the network address is already in use.
WSAEADDRNOTAVAIL	Indicates that the network address is not available.
WSAENETDOWN	Indicates that the network is down.
WSAENETUNREACH	Indicates that the network is unreachable.
WSAENETRESET	Indicates that the network connection has been reset.
WSAECONNABORTED	Indicates that the connection has been aborted.
WSAECONNRESET	Indicates that the connection has been reset by the peer.
WSAENOBUFS	Indicates that there is no buffer space available.
WSAEISCONN	Indicates that the socket is already connected.
WSAENOTCONN	Indicates that the socket is not connected.
WSAESHUTDOWN	Indicates that data cannot be sent after the socket has been shutdown.
WSAETOOMANYREFS	Indicates that there are too many references.
WSAETIMEDOUT	Indicates that the connection has timed out.
WSAECONNREFUSED	Indicates that the connection has been refused.
WSAELOOP	Indicates that a name cannot be translated.
WSAENAMETOOLONG	Indicates that a name was too long.
WSAEHOSTDOWN	Indicates that a network host is down.
WSAEHOSTUNREACH	Indicates that there is no route to a network host.
WSAENOTEMPTY	Indicates that the directory is not empty.
WSAEPROCLIM	Indicates that there are too many processes.
WSAEUSERS	Indicates that the user quota has been exceeded.
WSAEDQUOT	Indicates that the disk quota has been exceeded.
WSAESTALE	Indicates a stale file handle reference.
WSAEREMOTE	Indicates that the item is remote.
	Indicates that the network subsystem is not ready.

WSASYSNOTREADY	
WSAVERNOTSUPPORTED	Indicates that the winsock.dll version is out of range.
WSANOTINITIALISED	Indicates that successful WSAStartup has not yet been performed.
WSAEDISCON	Indicates that a graceful shutdown is in progress.
WSAENOMORE	Indicates that there are no more results.
WSAECANCELLED	Indicates that an operation has been canceled.
WSAEINVALIDPROCTABLE	Indicates that the procedure call table is invalid.
WSAEINVALIDPROVIDER	Indicates an invalid service provider.
WSAEPROVIDERFAILEDINIT	Indicates that the service provider failed to initialized.
WSASYSCALLFAILURE	Indicates a system call failure.
WSASERVICE_NOT_FOUND	Indicates that a service was not found.
WSATYPE_NOT_FOUND	Indicates that a class type was not found.
WSA_E_NO_MORE	Indicates that there are no more results.
WSA_E_CANCELLED	Indicates that the call was canceled.
WSAEREFUSED	Indicates that a database query was refused.

dlopen Constants

If available on the operating system, the following constants are exported in `os.constants.dlopen`. See dlopen(3) for detailed information.

Constant	Description
RTLD_LAZY	Perform lazy binding. Node.js sets this flag by default.
RTLD_NOW	Resolve all undefined symbols in the library before dlopen(3) returns.
RTLD_GLOBAL	Symbols defined by the library will be made available for symbol resolution of subsequently loaded libraries.
RTLD_LOCAL	The converse of RTLD_GLOBAL. This is the default behavior if neither flag is specified.
RTLD_DEEPBIND	Make a self-contained library use its own symbols in preference to symbols from previously loaded libraries.

libuv Constants

Constant	Description
UV_UDP_REUSEADDR	

Path

Stability: 2 - Stable

The `path` module provides utilities for working with file and directory paths. It can be accessed using:

```
const path = require('path');
```

Windows vs. POSIX

The default operation of the `path` module varies based on the operating system on which a Node.js application is running. Specifically, when running on a Windows operating system, the `path` module will assume that Windows-style paths are being used.

For example, using the `path.basename()` function with the Windows file path `C:\temp\myfile.html`, will yield different results when running on POSIX than when run on Windows:

On POSIX:

```
path.basename('C:\\temp\\myfile.html');
// Returns: 'C:\\temp\\myfile.html'
```

On Windows:

```
path.basename('C:\\temp\\myfile.html');
// Returns: 'myfile.html'
```

To achieve consistent results when working with Windows file paths on any operating system, use path.win32 :

On POSIX and Windows:

```
path.win32.basename('C:\\temp\\myfile.html');
// Returns: 'myfile.html'
```

To achieve consistent results when working with POSIX file paths on any operating system, use path.posix :

On POSIX and Windows:

```
path.posix.basename('/tmp/myfile.html');
// Returns: 'myfile.html'
```

Note: On Windows Node.js follows the concept of per-drive working directory. This behavior can be observed when using a drive path without a backslash. For example `path.resolve('c:\\')` can potentially return a different result than `path.resolve('c:')`. For more information, see this MSDN page.

path.basename(path[, ext])

▸ History

- `path` <string>
- `ext` <string> An optional file extension
- Returns: <string>

The `path.basename()` methods returns the last portion of a `path`, similar to the Unix `basename` command. Trailing directory separators are ignored, see path.sep .

```
path.basename('/foo/bar/baz/asdf/quux.html');
// Returns: 'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html');
// Returns: 'quux'
```

A TypeError is thrown if `path` is not a string or if `ext` is given and is not a string.

path.delimiter

Added in: v0.9.3

- <string>

Provides the platform-specific path delimiter:

- `;` for Windows
- `:` for POSIX

For example, on POSIX:

```
console.log(process.env.PATH);
// Prints: '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'

process.env.PATH.split(path.delimiter);
// Returns: ['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']
```

On Windows:

```
console.log(process.env.PATH);
// Prints: 'C:\Windows\system32;C:\Windows;C:\Program Files\node\'

process.env.PATH.split(path.delimiter);
// Returns ['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\node\\']
```

path.dirname(path)

▶ History

- `path` <string>
- Returns: <string>

The `path.dirname()` method returns the directory name of a `path`, similar to the Unix `dirname` command. Trailing directory separators are ignored, see path.sep .

```
path.dirname('/foo/bar/baz/asdf/quux');
// Returns: '/foo/bar/baz/asdf'
```

A TypeError is thrown if `path` is not a string.

path.extname(path)

▶ History

- `path` <string>
- Returns: <string>

The `path.extname()` method returns the extension of the `path`, from the last occurrence of the `.` (period) character to end of string in the last portion of the `path`. If there is no `.` in the last portion of the `path`, or if the first character of the basename of `path` (see `path.basename()`) is `.`, then an empty string is returned.

```
path.extname('index.html');
// Returns: '.html'

path.extname('index.coffee.md');
// Returns: '.md'

path.extname('index.');
// Returns: '.'

path.extname('index');
// Returns: ''

path.extname('.index');
// Returns: ''
```

A TypeError is thrown if `path` is not a string.

path.format(pathObject)

Added in: v0.11.15

- `pathObject` <Object>
  - `dir` <string>
  - `root` <string>
  - `base` <string>
  - `name` <string>
  - `ext` <string>
- Returns: <string>

The `path.format()` method returns a path string from an object. This is the opposite of path.parse().

When providing properties to the `pathObject` remember that there are combinations where one property has priority over another:

- `pathObject.root` is ignored if `pathObject.dir` is provided
- `pathObject.ext` and `pathObject.name` are ignored if `pathObject.base` exists

For example, on POSIX:

```
// If `dir`, `root` and `base` are provided,
// `${dir}${path.sep}${base}`
// will be returned. `root` is ignored.
path.format({
  root: '/ignored',
  dir: '/home/user/dir',
  base: 'file.txt'
});
// Returns: '/home/user/dir/file.txt'

// `root` will be used if `dir` is not specified.
// If only `root` is provided or `dir` is equal to `root` then the
// platform separator will not be included. `ext` will be ignored.
path.format({
  root: '/',
  base: 'file.txt',
  ext: 'ignored'
});
// Returns: '/file.txt'

// `name` + `ext` will be used if `base` is not specified.
path.format({
  root: '/',
  name: 'file',
  ext: '.txt'
});
// Returns: '/file.txt'
```

On Windows:

```
path.format({
  dir: 'C:\\path\\dir',
  base: 'file.txt'
});
// Returns: 'C:\\path\\dir\\file.txt'
```

path.isAbsolute(path)

Added in: v0.11.2

- `path` <string>
- Returns: <boolean>

The `path.isAbsolute()` method determines if `path` is an absolute path.

If the given `path` is a zero-length string, `false` will be returned.

For example on POSIX:

```
path.isAbsolute('/foo/bar'); // true
path.isAbsolute('/baz/..');  // true
path.isAbsolute('qux/');     // false
path.isAbsolute('.');        // false
```

On Windows:

```
path.isAbsolute('//server');    // true
path.isAbsolute('\\\\server');  // true
path.isAbsolute('C:/foo/..');   // true
path.isAbsolute('C:\\foo\\..'); // true
path.isAbsolute('bar\\baz');    // false
path.isAbsolute('bar/baz');     // false
path.isAbsolute('.');           // false
```

A TypeError is thrown if `path` is not a string.

path.join([...paths])

Added in: v0.1.16

- `...paths` <string> A sequence of path segments
- Returns: <string>

The `path.join()` method joins all given `path` segments together using the platform specific separator as a delimiter, then normalizes the resulting path.

Zero-length `path` segments are ignored. If the joined path string is a zero-length string then `'.'` will be returned, representing the current working directory.

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');
// Returns: '/foo/bar/baz/asdf'

path.join('foo', {}, 'bar');
// throws 'TypeError: Path must be a string. Received {}'
```

A TypeError is thrown if any of the path segments is not a string.

path.normalize(path)

Added in: v0.1.23

- `path` <string>
- Returns: <string>

The `path.normalize()` method normalizes the given `path`, resolving `'..'` and `'.'` segments.

When multiple, sequential path segment separation characters are found (e.g. `/` on POSIX and either `\` or `/` on Windows), they are replaced by a single instance of the platform specific path segment separator (`/` on POSIX and `\` on Windows). Trailing separators are preserved.

If the `path` is a zero-length string, `'.'` is returned, representing the current working directory.

For example on POSIX:

```
path.normalize('/foo/bar//baz/asdf/quux/..');
// Returns: '/foo/bar/baz/asdf'
```

On Windows:

```
path.normalize('C:\\temp\\\\foo\\bar\\..\\');
// Returns: 'C:\\temp\\foo\\'
```

Since Windows recognizes multiple path separators, both separators will be replaced by instances of the Windows preferred separator (`\`):

```
path.win32.normalize('C:////temp\\\\/\\/\\/foo/bar');
// Returns: 'C:\\temp\\foo\\bar'
```

A TypeError is thrown if `path` is not a string.

path.parse(path)

Added in: v0.11.15

- `path` <string>
- Returns: <Object>

The `path.parse()` method returns an object whose properties represent significant elements of the `path`. Trailing directory separators are ignored, see path.sep .

The returned object will have the following properties:

- `dir` <string>
- `root` <string>
- `base` <string>
- `name` <string>
- `ext` <string>

For example on POSIX:

```
path.parse('/home/user/dir/file.txt');
// Returns:
// { root: '/',
//   dir: '/home/user/dir',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file' }
```

```
┌─────────────────────┬────────────┐
│          dir        │    base    │
├──────┬              ├──────┬─────┤
│ root │              │ name │ ext │
"  /    home/user/dir / file  .txt "
└──────┴──────────────┴──────┴─────┘
(all spaces in the "" line should be ignored — they are purely for formatting)
```

On Windows:

```
path.parse('C:\\path\\dir\\file.txt');
// Returns:
// { root: 'C:\\',
//   dir: 'C:\\path\\dir',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file' }
```

```
┌─────────────────────┬──────────────┐
│          dir        │    base      │
├──────┬              ├──────┬───────┤
│ root │              │ name │ ext   │
" C:\      path\dir   \ file  .txt "
└──────┴──────────────┴──────┴───────┘
(all spaces in the "" line should be ignored — they are purely for formatting)
```

A TypeError is thrown if `path` is not a string.

path.posix

Added in: v0.11.15

- <Object>

The `path.posix` property provides access to POSIX specific implementations of the `path` methods.

path.relative(from, to)

▸ History

- `from` <string>
- `to` <string>
- Returns: <string>

The `path.relative()` method returns the relative path from `from` to `to` based on the current working directory. If `from` and `to` each resolve to the same path (after calling `path.resolve()` on each), a zero-length string is returned.

If a zero-length string is passed as `from` or `to`, the current working directory will be used instead of the zero-length strings.

For example on POSIX:

```
path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb');
// Returns: '../../impl/bbb'
```

On Windows:

```
path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb');
// Returns: '..\\..\\impl\\bbb'
```

A TypeError is thrown if either `from` or `to` is not a string.

path.resolve([...paths])

Added in: v0.3.4

- `...paths` <string> A sequence of paths or path segments
- Returns: <string>

The `path.resolve()` method resolves a sequence of paths or path segments into an absolute path.

The given sequence of paths is processed from right to left, with each subsequent `path` prepended until an absolute path is constructed. For instance, given the sequence of path segments: `/foo`, `/bar`, `baz`, calling `path.resolve('/foo', '/bar', 'baz')` would return `/bar/baz`.

If after processing all given `path` segments an absolute path has not yet been generated, the current working directory is used.

The resulting path is normalized and trailing slashes are removed unless the path is resolved to the root directory.

Zero-length `path` segments are ignored.

If no `path` segments are passed, `path.resolve()` will return the absolute path of the current working directory.

```
path.resolve('/foo/bar', './baz');
// Returns: '/foo/bar/baz'

path.resolve('/foo/bar', '/tmp/file/');
// Returns: '/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif');
// if the current working directory is /home/myself/node,
// this returns '/home/myself/node/wwwroot/static_files/gif/image.gif'
```

A TypeError is thrown if any of the arguments is not a string.

path.sep

Added in: v0.7.9

- <string>

Provides the platform-specific path segment separator:

- `\` on Windows
- `/` on POSIX

For example on POSIX:

```
'foo/bar/baz'.split(path.sep);
// Returns: ['foo', 'bar', 'baz']
```

On Windows:

```
'foo\\bar\\baz'.split(path.sep);
// Returns: ['foo', 'bar', 'baz']
```

On Windows, both the forward slash (`/`) and backward slash (`\`) are accepted as path segment separators; however, the `path` methods only add backward slashes (`\`).

path.toNamespacedPath(path)

Added in: v9.0.0

- `path` <string>
- Returns: <string>

On Windows systems only, returns an equivalent namespace-prefixed path for the given `path`. If `path` is not a string, `path` will be returned without modifications.

This method is meaningful only on Windows system. On posix systems, the method is non-operational and always returns `path` without modifications.

path.win32

Added in: v0.11.15

- <Object>

The `path.win32` property provides access to Windows-specific implementations of the `path` methods.

Performance Timing API

Stability: 1 - Experimental

The Performance Timing API provides an implementation of the W3C Performance Timeline specification. The purpose of the API is to support collection of high resolution performance metrics. This is the same Performance API as implemented in modern Web browsers.

```
const { performance } = require('perf_hooks');
performance.mark('A');
doSomeLongRunningProcess(() => {
  performance.mark('B');
  performance.measure('A to B', 'A', 'B');
  const measure = performance.getEntriesByName('A to B')[0];
  console.log(measure.duration);
  // Prints the number of milliseconds between Mark 'A' and Mark 'B'
});
```

Class: Performance

Added in: v8.5.0

The `Performance` provides access to performance metric data. A single instance of this class is provided via the `performance` property.

performance.clearEntries(name)

Added in: v9.5.0

Remove all performance entry objects with `entryType` equal to `name` from the Performance Timeline.

performance.clearFunctions([name])

Added in: v8.5.0

- `name` <string>

If `name` is not provided, removes all `PerformanceFunction` objects from the Performance Timeline. If `name` is provided, removes entries with `name`.

performance.clearGC()

Added in: v8.5.0

Remove all performance entry objects with `entryType` equal to `gc` from the Performance Timeline.

performance.clearMarks([name])

Added in: v8.5.0

- `name` <string>

If `name` is not provided, removes all `PerformanceMark` objects from the Performance Timeline. If `name` is provided, removes only the named mark.

performance.clearMeasures([name])

Added in: v8.5.0

- `name` <string>

If `name` is not provided, removes all `PerformanceMeasure` objects from the Performance Timeline. If `name` is provided, removes only objects whose `performanceEntry.name` matches `name`.

performance.getEntries()

Added in: v8.5.0

- Returns: <Array>

Returns a list of all `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime`.

performance.getEntriesByName(name[, type])

Added in: v8.5.0

- `name` <string>
- `type` <string>
- Returns: <Array>

Returns a list of all `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.name` is equal to `name`, and optionally, whose `performanceEntry.entryType` is equal to `type`.

performance.getEntriesByType(type)

Added in: v8.5.0

- `type` <string>
- Returns: <Array>

Returns a list of all `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.entryType` is equal to `type`.

performance.mark([name])

Added in: v8.5.0

- `name` <string>

Creates a new `PerformanceMark` entry in the Performance Timeline. A `PerformanceMark` is a subclass of `PerformanceEntry` whose `performanceEntry.entryType` is always `'mark'`, and whose `performanceEntry.duration` is always `0`. Performance marks are used to mark specific significant moments in the Performance Timeline.

performance.maxEntries

Added in: v9.6.0

Value: <number>

The maximum number of Performance Entry items that should be added to the Performance Timeline. This limit is not strictly enforced, but a process warning will be emitted if the number of entries in the timeline exceeds this limit.

Defaults to 150.

performance.measure(name, startMark, endMark)

Added in: v8.5.0

- `name` <string>
- `startMark` <string>
- `endMark` <string>

Creates a new `PerformanceMeasure` entry in the Performance Timeline. A `PerformanceMeasure` is a subclass of `PerformanceEntry` whose `performanceEntry.entryType` is always `'measure'`, and whose `performanceEntry.duration` measures the number of milliseconds elapsed since `startMark` and `endMark`.

The `startMark` argument may identify any *existing* `PerformanceMark` in the Performance Timeline, or *may* identify any of the timestamp properties provided by the `PerformanceNodeTiming` class. If the named `startMark` does not exist, then `startMark` is set to timeOrigin by default.

The `endMark` argument must identify any *existing* `PerformanceMark` in the Performance Timeline or any of the timestamp properties provided by the `PerformanceNodeTiming` class. If the named `endMark` does not exist, an error will be thrown.

performance.nodeTiming

Added in: v8.5.0

- <PerformanceNodeTiming>

An instance of the `PerformanceNodeTiming` class that provides performance metrics for specific Node.js operational milestones.

performance.now()

Added in: v8.5.0

- Returns: <number>

Returns the current high resolution millisecond timestamp, where 0 represents the start of the current `node` process.

performance.timeOrigin

Added in: v8.5.0

- <number>

The `timeOrigin` specifies the high resolution millisecond timestamp at which the current `node` process began, measured in Unix time.

performance.timerify(fn)

Added in: v8.5.0

- `fn` <Function>

Wraps a function within a new function that measures the running time of the wrapped function. A `PerformanceObserver` must be subscribed to the `'function'` event type in order for the timing details to be accessed.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

function someFunction() {
  console.log('hello world');
}

const wrapped = performance.timerify(someFunction);

const obs = new PerformanceObserver((list) => {
  console.log(list.getEntries()[0].duration);
  obs.disconnect();
  performance.clearFunctions();
});
obs.observe({ entryTypes: ['function'] });

// A performance timeline entry will be created
wrapped();
```

Class: PerformanceEntry

Added in: v8.5.0

performanceEntry.duration

Added in: v8.5.0

- <number>

The total number of milliseconds elapsed for this entry. This value will not be meaningful for all Performance Entry types.

performanceEntry.name

Added in: v8.5.0

- <string>

The name of the performance entry.

performanceEntry.startTime

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp marking the starting time of the Performance Entry.

performanceEntry.entryType

Added in: v8.5.0

- <string>

The type of the performance entry. Current it may be one of: `'node'`, `'mark'`, `'measure'`, `'gc'`, or `'function'`.

performanceEntry.kind

Added in: v8.5.0

- <number>

When `performanceEntry.entryType` is equal to `'gc'`, the `performance.kind` property identifies the type of garbage collection operation that occurred. The value may be one of:

- `perf_hooks.constants.NODE_PERFORMANCE_GC_MAJOR`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_MINOR`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_INCREMENTAL`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_WEAKCB`

Class: PerformanceNodeTiming extends PerformanceEntry

Added in: v8.5.0

Provides timing details for Node.js itself.

performanceNodeTiming.bootstrapComplete

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which the Node.js process completed bootstrapping. If bootstrapping has not yet finished, the property has the value of -1.

performanceNodeTiming.clusterSetupEnd

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which cluster processing ended. If cluster processing has not yet ended, the property has the value of -1.

performanceNodeTiming.clusterSetupStart

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which cluster processing started. If cluster processing has not yet started, the property has the value of -1.

performanceNodeTiming.loopExit

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which the Node.js event loop exited. If the event loop has not yet exited, the property has the value of -1. It can only have a value of not -1 in a handler of the `'exit'` event.

performanceNodeTiming.loopStart

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which the Node.js event loop started. If the event loop has not yet started (e.g., in the first tick of the main script), the property has the value of -1.

performanceNodeTiming.moduleLoadEnd

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which main module load ended.

performanceNodeTiming.moduleLoadStart

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which main module load started.

performanceNodeTiming.nodeStart

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which the Node.js process was initialized.

performanceNodeTiming.preloadModuleLoadEnd

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which preload module load ended.

performanceNodeTiming.preloadModuleLoadStart

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which preload module load started.

performanceNodeTiming.thirdPartyMainEnd

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which third_party_main processing ended. If third_party_main processing has not yet ended, the property has the value of -1.

performanceNodeTiming.thirdPartyMainStart

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which third_party_main processing started. If third_party_main processing has not yet started, the property has the value of -1.

performanceNodeTiming.v8Start

Added in: v8.5.0

- <number>

The high resolution millisecond timestamp at which the V8 platform was initialized.

Class: PerformanceObserver(callback)

Added in: v8.5.0

- `callback` <Function> A `PerformanceObserverCallback` callback function.

`PerformanceObserver` objects provide notifications when new `PerformanceEntry` instances have been added to the Performance Timeline.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((list, observer) => {
  console.log(list.getEntries());
  observer.disconnect();
});
obs.observe({ entryTypes: ['mark'], buffered: true });

performance.mark('test');
```

Because `PerformanceObserver` instances introduce their own additional performance overhead, instances should not be left subscribed to notifications indefinitely. Users should disconnect observers as soon as they are no longer needed.

Callback: PerformanceObserverCallback(list, observer)

Added in: v8.5.0

- `list` <PerformanceObserverEntryList>
- `observer` <PerformanceObserver>

The `PerformanceObserverCallback` is invoked when a `PerformanceObserver` is notified about new `PerformanceEntry` instances. The callback receives a `PerformanceObserverEntryList` instance and a reference to the `PerformanceObserver`.

Class: PerformanceObserverEntryList

Added in: v8.5.0

The `PerformanceObserverEntryList` class is used to provide access to the `PerformanceEntry` instances passed to a `PerformanceObserver`.

performanceObserverEntryList.getEntries()

Added in: v8.5.0

- Returns: <Array>

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime`.

performanceObserverEntryList.getEntriesByName(name[, type])

Added in: v8.5.0

- `name` <string>
- `type` <string>
- Returns: <Array>

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.name` is equal to `name`, and optionally, whose `performanceEntry.entryType` is equal to `type`.

performanceObserverEntryList.getEntriesByType(type)

Added in: v8.5.0

- `type` <string>
- Returns: <Array>

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.entryType` is equal to `type`.

performanceObserver.disconnect()

Added in: v8.5.0

Disconnects the `PerformanceObserver` instance from all notifications.

performanceObserver.observe(options)

Added in: v8.5.0

- `options` <Object>
  - `entryTypes` <Array> An array of strings identifying the types of `PerformanceEntry` instances the observer is interested in. If not provided an error will be thrown.
  - `buffered` <boolean> If true, the notification callback will be called using `setImmediate()` and multiple `PerformanceEntry` instance notifications will be buffered internally. If `false`, notifications will be immediate and synchronous. Defaults to `false`.

Subscribes the `PerformanceObserver` instance to notifications of new `PerformanceEntry` instances identified by `options.entryTypes`.

When `options.buffered` is `false`, the `callback` will be invoked once for every `PerformanceEntry` instance:

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((list, observer) => {
  // called three times synchronously. list contains one item
});
obs.observe({ entryTypes: ['mark'] });

for (let n = 0; n < 3; n++)
  performance.mark(`test${n}`);
```

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((list, observer) => {
  // called once. list contains three items
});
obs.observe({ entryTypes: ['mark'], buffered: true });

for (let n = 0; n < 3; n++)
  performance.mark(`test${n}`);
```

Examples

Measuring the duration of async operations

The following example uses the Async Hooks and Performance APIs to measure the actual duration of a Timeout operation (including the amount of time it to execute the callback).

```
'use strict';
const async_hooks = require('async_hooks');
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const set = new Set();
const hook = async_hooks.createHook({
  init(id, type) {
    if (type === 'Timeout') {
      performance.mark(`Timeout-${id}-Init`);
      set.add(id);
    }
  },
  destroy(id) {
    if (set.has(id)) {
      set.delete(id);
      performance.mark(`Timeout-${id}-Destroy`);
      performance.measure(`Timeout-${id}`,
                          `Timeout-${id}-Init`,
                          `Timeout-${id}-Destroy`);
    }
  }
});
hook.enable();

const obs = new PerformanceObserver((list, observer) => {
  console.log(list.getEntries()[0]);
  performance.clearMarks();
  performance.clearMeasures();
  observer.disconnect();
});
obs.observe({ entryTypes: ['measure'], buffered: true });

setTimeout(() => {}, 1000);
```

Measuring how long it takes to load dependencies

The following example measures the duration of `require()` operations to load dependencies:

```
'use strict';
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');
const mod = require('module');

// Monkey patch the require function
mod.Module.prototype.require =
  performance.timerify(mod.Module.prototype.require);
require = performance.timerify(require);

// Activate the observer
const obs = new PerformanceObserver((list) => {
  const entries = list.getEntries();
  entries.forEach((entry) => {
    console.log(`require('${entry[0]}')`, entry.duration);
  });
  obs.disconnect();
  // Free memory
  performance.clearFunctions();
});
obs.observe({ entryTypes: ['function'], buffered: true });

require('some-module');
```

Process

The `process` object is a `global` that provides information about, and control over, the current Node.js process. As a global, it is always available to Node.js applications without using `require()`.

Process Events

The `process` object is an instance of EventEmitter.

Event: 'beforeExit'

Added in: v0.11.12

The `'beforeExit'` event is emitted when Node.js empties its event loop and has no additional work to schedule. Normally, the Node.js process will exit when there is no work scheduled, but a listener registered on the `'beforeExit'` event can make asynchronous calls, and thereby cause the Node.js process to continue.

The listener callback function is invoked with the value of process.exitCode passed as the only argument.

The `'beforeExit'` event is *not* emitted for conditions causing explicit termination, such as calling process.exit() or uncaught exceptions.

The `'beforeExit'` should *not* be used as an alternative to the `'exit'` event unless the intention is to schedule additional work.

Event: 'disconnect'

Added in: v0.7.7

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the `'disconnect'` event will be emitted when the IPC channel is closed.

Event: 'exit'

Added in: v0.1.7

The 'exit' event is emitted when the Node.js process is about to exit as a result of either:

- The process.exit() method being called explicitly;
- The Node.js event loop no longer having any additional work to perform.

There is no way to prevent the exiting of the event loop at this point, and once all 'exit' listeners have finished running the Node.js process will terminate.

The listener callback function is invoked with the exit code specified either by the process.exitCode property, or the exitCode argument passed to the process.exit() method, as the only argument.

```
process.on('exit', (code) => {
  console.log(`About to exit with code: ${code}`);
});
```

Listener functions **must** only perform **synchronous** operations. The Node.js process will exit immediately after calling the 'exit' event listeners causing any additional work still queued in the event loop to be abandoned. In the following example, for instance, the timeout will never occur:

```
process.on('exit', (code) => {
  setTimeout(() => {
    console.log('This will not run');
  }, 0);
});
```

Event: 'message'

Added in: v0.5.10

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the 'message' event is emitted whenever a message sent by a parent process using childprocess.send() is received by the child process.

The listener callback is invoked with the following arguments:

- message <Object> a parsed JSON object or primitive value.
- sendHandle <Handle object> a net.Socket or net.Server object, or undefined.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

Event: 'rejectionHandled'

Added in: v1.4.1

The 'rejectionHandled' event is emitted whenever a Promise has been rejected and an error handler was attached to it (using promise.catch(), for example) later than one turn of the Node.js event loop.

The listener callback is invoked with a reference to the rejected Promise as the only argument.

The Promise object would have previously been emitted in an 'unhandledRejection' event, but during the course of processing gained a rejection handler.

There is no notion of a top level for a Promise chain at which rejections can always be handled. Being inherently asynchronous in nature, a Promise rejection can be handled at a future point in time — possibly much later than the event loop turn it takes for the 'unhandledRejection' event to be emitted.

Another way of stating this is that, unlike in synchronous code where there is an ever-growing list of unhandled exceptions, with Promises there can be a growing-and-shrinking list of unhandled rejections.

In synchronous code, the 'uncaughtException' event is emitted when the list of unhandled exceptions grows.

In asynchronous code, the `'unhandledRejection'` event is emitted when the list of unhandled rejections grows, and the `'rejectionHandled'` event is emitted when the list of unhandled rejections shrinks.

```
const unhandledRejections = new Map();
process.on('unhandledRejection', (reason, p) => {
  unhandledRejections.set(p, reason);
});
process.on('rejectionHandled', (p) => {
  unhandledRejections.delete(p);
});
```

In this example, the `unhandledRejections` `Map` will grow and shrink over time, reflecting rejections that start unhandled and then become handled. It is possible to record such errors in an error log, either periodically (which is likely best for long-running application) or upon process exit (which is likely most convenient for scripts).

Event: 'uncaughtException'

Added in: v0.1.18

The `'uncaughtException'` event is emitted when an uncaught JavaScript exception bubbles all the way back to the event loop. By default, Node.js handles such exceptions by printing the stack trace to `stderr` and exiting. Adding a handler for the `'uncaughtException'` event overrides this default behavior.

The listener function is called with the `Error` object passed as the only argument.

```
process.on('uncaughtException', (err) => {
  fs.writeSync(1, `Caught exception: ${err}\n`);
});

setTimeout(() => {
  console.log('This will still run.');
}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

Warning: Using `'uncaughtException'` correctly

Note that `'uncaughtException'` is a crude mechanism for exception handling intended to be used only as a last resort. The event*should not* be used as an equivalent to `On Error Resume Next`. Unhandled exceptions inherently mean that an application is in an undefined state. Attempting to resume application code without properly recovering from the exception can cause additional unforeseen and unpredictable issues.

Exceptions thrown from within the event handler will not be caught. Instead the process will exit with a non-zero exit code and the stack trace will be printed. This is to avoid infinite recursion.

Attempting to resume normally after an uncaught exception can be similar to pulling out of the power cord when upgrading a computer — nine out of ten times nothing happens - but the 10th time, the system becomes corrupted.

The correct use of `'uncaughtException'` is to perform synchronous cleanup of allocated resources (e.g. file descriptors, handles, etc) before shutting down the process. **It is not safe to resume normal operation after `'uncaughtException'`.**

To restart a crashed application in a more reliable way, whether `uncaughtException` is emitted or not, an external monitor should be employed in a separate process to detect application failures and recover or restart as needed.

Event: 'unhandledRejection'

▸ History

The 'unhandledRejection' event is emitted whenever a Promise is rejected and no error handler is attached to the promise within a turn of the event loop. When programming with Promises, exceptions are encapsulated as "rejected promises". Rejections can be caught and handled using promise.catch() and are propagated through a Promise chain. The 'unhandledRejection' event is useful for detecting and keeping track of promises that were rejected whose rejections have not yet been handled.

The listener function is called with the following arguments:

- reason <Error> | <any> The object with which the promise was rejected (typically an Error object).
- p the Promise that was rejected.

```
process.on('unhandledRejection', (reason, p) => {
  console.log('Unhandled Rejection at:', p, 'reason:', reason);
  // application specific logging, throwing an error, or other logic here
});

somePromise.then((res) => {
  return reportToUser(JSON.pasre(res)); // note the typo (`pasre`)
}); // no `.catch` or `.then`
```

The following will also trigger the 'unhandledRejection' event to be emitted:

```
function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}

const resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

In this example case, it is possible to track the rejection as a developer error as would typically be the case for other 'unhandledRejection' events. To address such failures, a non-operational .catch(() => { }) handler may be attached to resource.loaded, which would prevent the 'unhandledRejection' event from being emitted. Alternatively, the 'rejectionHandled' event may be used.

Event: 'warning'

Added in: v6.0.0

The 'warning' event is emitted whenever Node.js emits a process warning.

A process warning is similar to an error in that it describes exceptional conditions that are being brought to the user's attention. However, warnings are not part of the normal Node.js and JavaScript error handling flow. Node.js can emit warnings whenever it detects bad coding practices that could lead to sub-optimal application performance, bugs, or security vulnerabilities.

The listener function is called with a single warning argument whose value is an Error object. There are three key properties that describe the warning:

- name <string> The name of the warning (currently Warning by default).
- message <string> A system-provided description of the warning.
- stack <string> A stack trace to the location in the code where the warning was issued.

```
process.on('warning', (warning) => {
  console.warn(warning.name);    // Print the warning name
  console.warn(warning.message); // Print the warning message
  console.warn(warning.stack);   // Print the stack trace
});
```

By default, Node.js will print process warnings to stderr. The --no-warnings command-line option can be used to suppress the default console

output but the `'warning'` event will still be emitted by the `process` object.

The following example illustrates the warning that is printed to `stderr` when too many listeners have been added to an event

```
$ node
> events.defaultMaxListeners = 1;
> process.on('foo', () => {});
> process.on('foo', () => {});
> (node:38638) MaxListenersExceededWarning: Possible EventEmitter memory leak
detected. 2 foo listeners added. Use emitter.setMaxListeners() to increase limit
```

In contrast, the following example turns off the default warning output and adds a custom handler to the `'warning'` event:

```
$ node --no-warnings
> const p = process.on('warning', (warning) => console.warn('Do not do that!'));
> events.defaultMaxListeners = 1;
> process.on('foo', () => {});
> process.on('foo', () => {});
> Do not do that!
```

The `--trace-warnings` command-line option can be used to have the default console output for warnings include the full stack trace of the warning.

Launching Node.js using the `--throw-deprecation` command line flag will cause custom deprecation warnings to be thrown as exceptions.

Using the `--trace-deprecation` command line flag will cause the custom deprecation to be printed to `stderr` along with the stack trace.

Using the `--no-deprecation` command line flag will suppress all reporting of the custom deprecation.

The `*-deprecation` command line flags only affect warnings that use the name `DeprecationWarning`.

Emitting custom warnings

See the `process.emitWarning()` method for issuing custom or application-specific warnings.

Signal Events

Signal events will be emitted when the Node.js process receives a signal. Please refer to signal(7) for a listing of standard POSIX signal names such as `SIGINT`, `SIGHUP`, etc.

The signal handler will receive the signal's name (`'SIGINT'`, `'SIGTERM'`, etc.) as the first argument.

The name of each event will be the uppercase common name for the signal (e.g. `'SIGINT'` for `SIGINT` signals).

```
// Begin reading from stdin so the process does not exit.
process.stdin.resume();

process.on('SIGINT', () => {
  console.log('Received SIGINT. Press Control-D to exit.');
});

// Using a single function to handle multiple signals
function handle(signal) {
  console.log(`Received ${signal}`);
}

process.on('SIGINT', handle);
process.on('SIGTERM', handle);
```

- `SIGUSR1` is reserved by Node.js to start the debugger . It's possible to install a listener but doing so might interfere with the debugger.
- `SIGTERM` and `SIGINT` have default handlers on non-Windows platforms that reset the terminal mode before exiting with code `128 + signal number`. If one of these signals has a listener installed, its default behavior will be removed (Node.js will no longer exit).
- `SIGPIPE` is ignored by default. It can have a listener installed.
- `SIGHUP` is generated on Windows when the console window is closed, and on other platforms under various similar conditions, see signal(7) . It can have a listener installed, however Node.js will be unconditionally terminated by Windows about 10 seconds later. On non-Windows platforms, the default behavior of `SIGHUP` is to terminate Node.js, but once a listener has been installed its default behavior will be removed.
- `SIGTERM` is not supported on Windows, it can be listened on.
- `SIGINT` from the terminal is supported on all platforms, and can usually be generated with `<Ctrl>+C` (though this may be configurable). It is not generated when terminal raw mode is enabled.
- `SIGBREAK` is delivered on Windows when `<Ctrl>+<Break>` is pressed, on non-Windows platforms it can be listened on, but there is no way to send or generate it.
- `SIGWINCH` is delivered when the console has been resized. On Windows, this will only happen on write to the console when the cursor is being moved, or when a readable tty is used in raw mode.
- `SIGKILL` cannot have a listener installed, it will unconditionally terminate Node.js on all platforms.
- `SIGSTOP` cannot have a listener installed.
- `SIGBUS` , `SIGFPE` , `SIGSEGV` and `SIGILL` , when not raised artificially using kill(2) , inherently leave the process in a state from which it is not safe to attempt to call JS listeners. Doing so might lead to the process hanging in an endless loop, since listeners attached using `process.on()` are called asynchronously and therefore unable to correct the underlying problem.

Windows does not support sending signals, but Node.js offers some emulation with `process.kill()` , and `subprocess.kill()` . Sending signal `0` can be used to test for the existence of a process. Sending `SIGINT` , `SIGTERM` , and `SIGKILL` cause the unconditional termination of the target process.

process.abort()

Added in: v0.7.0

The `process.abort()` method causes the Node.js process to exit immediately and generate a core file.

process.arch

Added in: v0.5.0

- <string>

The `process.arch` property returns a string identifying the operating system CPU architecture for which the Node.js binary was compiled.

The current possible values are: `'arm'` , `'arm64'` , `'ia32'` , `'mips'` , `'mipsel'` , `'ppc'` , `'ppc64'` , `'s390'` , `'s390x'` , `'x32'` , and `'x64'` .

```
console.log(`This processor architecture is ${process.arch}`);
```

process.argv

Added in: v0.1.27

- <Array>

The `process.argv` property returns an array containing the command line arguments passed when the Node.js process was launched. The first element will be process.execPath. See `process.argv0` if access to the original value of `argv[0]` is needed. The second element will be the path to the JavaScript file being executed. The remaining elements will be any additional command line arguments.

For example, assuming the following script for `process-args.js`:

```
// print process.argv
process.argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});
```

Launching the Node.js process as:

```
$ node process-args.js one two=three four
```

Would generate the output:

```
0: /usr/local/bin/node
1: /Users/mjr/work/node/process-args.js
2: one
3: two=three
4: four
```

process.argv0

Added in: v6.4.0

- <string>

The `process.argv0` property stores a read-only copy of the original value of `argv[0]` passed when Node.js starts.

```
$ bash -c 'exec -a customArgv0 ./node'
> process.argv[0]
'/Volumes/code/external/node/out/Release/node'
> process.argv0
'customArgv0'
```

process.channel

Added in: v7.1.0

- <Object>

If the Node.js process was spawned with an IPC channel (see the Child Process documentation), the `process.channel` property is a reference to the IPC channel. If no IPC channel exists, this property is `undefined`.

process.chdir(directory)

Added in: v0.1.17

- `directory` <string>

The `process.chdir()` method changes the current working directory of the Node.js process or throws an exception if doing so fails (for instance,

if the specified directory does not exist).

```
console.log(`Starting directory: ${process.cwd()}`);
try {
  process.chdir('/tmp');
  console.log(`New directory: ${process.cwd()}`);
} catch (err) {
  console.error(`chdir: ${err}`);
}
```

process.config

Added in: v0.7.7

- <Object>

The process.config property returns an Object containing the JavaScript representation of the configure options used to compile the current Node.js executable. This is the same as the config.gypi file that was produced when running the ./configure script.

An example of the possible output looks like:

```
{
  target_defaults:
   { cflags: [],
     default_configuration: 'Release',
     defines: [],
     include_dirs: [],
     libraries: [] },
  variables:
   {
     host_arch: 'x64',
     node_install_npm: 'true',
     node_prefix: '',
     node_shared_cares: 'false',
     node_shared_http_parser: 'false',
     node_shared_libuv: 'false',
     node_shared_zlib: 'false',
     node_use_dtrace: 'false',
     node_use_openssl: 'true',
     node_shared_openssl: 'false',
     strict_aliasing: 'true',
     target_arch: 'x64',
     v8_use_snapshot: 'true'
   }
}
```

The process.config property is **not** read-only and there are existing modules in the ecosystem that are known to extend, modify, or entirely replace the value of process.config.

process.connected

Added in: v0.7.2

- <boolean>

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the process.connected property will return true so long as the IPC channel is connected and will return false after process.disconnect() is called.

Once process.connected is false, it is no longer possible to send messages over the IPC channel using process.send().

process.cpuUsage([previousValue])

Added in: v6.1.0

- `previousValue` <Object> A previous return value from calling `process.cpuUsage()`
- Returns: <Object>
  - `user` <integer>
  - `system` <integer>

The `process.cpuUsage()` method returns the user and system CPU time usage of the current process, in an object with properties `user` and `system`, whose values are microsecond values (millionth of a second). These values measure time spent in user and system code respectively, and may end up being greater than actual elapsed time if multiple CPU cores are performing work for this process.

The result of a previous call to `process.cpuUsage()` can be passed as the argument to the function, to get a diff reading.

```
const startUsage = process.cpuUsage();
// { user: 38579, system: 6986 }

// spin the CPU for 500 milliseconds
const now = Date.now();
while (Date.now() - now < 500);

console.log(process.cpuUsage(startUsage));
// { user: 514883, system: 11226 }
```

process.cwd()

Added in: v0.1.8

- Returns: <string>

The `process.cwd()` method returns the current working directory of the Node.js process.

```
console.log(`Current directory: ${process.cwd()}`);
```

process.debugPort

Added in: v0.7.2

- <number>

The port used by Node.js's debugger when enabled.

```
process.debugPort = 5858;
```

process.disconnect()

Added in: v0.7.2

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the `process.disconnect()` method will close the IPC channel to the parent process, allowing the child process to exit gracefully once there are no other connections keeping it alive.

The effect of calling `process.disconnect()` is that same as calling the parent process's `ChildProcess.disconnect()` .

If the Node.js process was not spawned with an IPC channel, `process.disconnect()` will be `undefined`.

process.dlopen(module, filename[, flags])

▸ History

- `module` <Object>
- `filename` <string>
- `flags` <os.constants.dlopen> Defaults to `os.constants.dlopen.RTLD_LAZY`.

The `process.dlopen()` method allows to dynamically load shared objects. It is primarily used by `require()` to load C++ Addons, and should not be used directly, except in special cases. In other words, `require()` should be preferred over `process.dlopen()`, unless there are specific reasons.

The `flags` argument is an integer that allows to specify dlopen behavior. See the `os.constants.dlopen` documentation for details.

If there are specific reasons to use `process.dlopen()` (for instance, to specify dlopen flags), it's often useful to use `require.resolve()` to look up the module's path.

An important drawback when calling `process.dlopen()` is that the `module` instance must be passed. Functions exported by the C++ Addon will be accessible via `module.exports`.

The example below shows how to load a C++ Addon, named as `binding`, that exports a `foo` function. All the symbols will be loaded before the call returns, by passing the `RTLD_NOW` constant. In this example the constant is assumed to be available.

```
const os = require('os');
process.dlopen(module, require.resolve('binding'),
               os.constants.dlopen.RTLD_NOW);
module.exports.foo();
```

process.emitWarning(warning[, options])

Added in: v8.0.0

- `warning` <string> | <Error> The warning to emit.
- `options` <Object>
  - `type` <string> When `warning` is a String, `type` is the name to use for the *type* of warning being emitted. Default: `Warning`.
  - `code` <string> A unique identifier for the warning instance being emitted.
  - `ctor` <Function> When `warning` is a String, `ctor` is an optional function used to limit the generated stack trace. Default `process.emitWarning`
  - `detail` <string> Additional text to include with the error.

The `process.emitWarning()` method can be used to emit custom or application specific process warnings. These can be listened for by adding a handler to the `process.on('warning')` event.

```
// Emit a warning with a code and additional detail.
process.emitWarning('Something happened!', {
  code: 'MY_WARNING',
  detail: 'This is some additional information'
});
// Emits:
// (node:56338) [MY_WARNING] Warning: Something happened!
// This is some additional information
```

In this example, an `Error` object is generated internally by `process.emitWarning()` and passed through to the `process.on('warning')` event.

```
process.on('warning', (warning) => {
  console.warn(warning.name);    // 'Warning'
  console.warn(warning.message); // 'Something happened!'
  console.warn(warning.code);    // 'MY_WARNING'
  console.warn(warning.stack);   // Stack trace
  console.warn(warning.detail);  // 'This is some additional information'
});
```

If `warning` is passed as an `Error` object, the `options` argument is ignored.

process.emitWarning(warning[, type[, code]][, ctor])

Added in: v6.0.0

- `warning` <string> | <Error> The warning to emit.
- `type` <string> When `warning` is a String, `type` is the name to use for the *type* of warning being emitted. Default: `Warning`.
- `code` <string> A unique identifier for the warning instance being emitted.
- `ctor` <Function> When `warning` is a String, `ctor` is an optional function used to limit the generated stack trace. Default `process.emitWarning`

The `process.emitWarning()` method can be used to emit custom or application specific process warnings. These can be listened for by adding a handler to the `process.on('warning')` event.

```
// Emit a warning using a string.
process.emitWarning('Something happened!');
// Emits: (node: 56338) Warning: Something happened!
```

```
// Emit a warning using a string and a type.
process.emitWarning('Something Happened!', 'CustomWarning');
// Emits: (node:56338) CustomWarning: Something Happened!
```

```
process.emitWarning('Something happened!', 'CustomWarning', 'WARN001');
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!
```

In each of the previous examples, an `Error` object is generated internally by `process.emitWarning()` and passed through to the `process.on('warning')` event.

```
process.on('warning', (warning) => {
  console.warn(warning.name);
  console.warn(warning.message);
  console.warn(warning.code);
  console.warn(warning.stack);
});
```

If `warning` is passed as an `Error` object, it will be passed through to the `process.on('warning')` event handler unmodified (and the optional `type`, `code` and `ctor` arguments will be ignored):

```
// Emit a warning using an Error object.
const myWarning = new Error('Something happened!');
// Use the Error name property to specify the type name
myWarning.name = 'CustomWarning';
myWarning.code = 'WARN001';

process.emitWarning(myWarning);
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!
```

A `TypeError` is thrown if `warning` is anything other than a string or `Error` object.

Note that while process warnings use `Error` objects, the process warning mechanism is **not** a replacement for normal error handling mechanisms.

The following additional handling is implemented if the warning `type` is `DeprecationWarning`:

- If the `--throw-deprecation` command-line flag is used, the deprecation warning is thrown as an exception rather than being emitted as an event.
- If the `--no-deprecation` command-line flag is used, the deprecation warning is suppressed.
- If the `--trace-deprecation` command-line flag is used, the deprecation warning is printed to `stderr` along with the full stack trace.

Avoiding duplicate warnings

As a best practice, warnings should be emitted only once per process. To do so, it is recommended to place the `emitWarning()` behind a simple boolean flag as illustrated in the example below:

```
function emitMyWarning() {
  if (!emitMyWarning.warned) {
    emitMyWarning.warned = true;
    process.emitWarning('Only warn once!');
  }
}
emitMyWarning();
// Emits: (node: 56339) Warning: Only warn once!
emitMyWarning();
// Emits nothing
```

process.env

Added in: v0.1.27

- <Object>

The `process.env` property returns an object containing the user environment. See environron(7) .

An example of this object looks like:

```
{
  TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node'
}
```

It is possible to modify this object, but such modifications will not be reflected outside the Node.js process. In other words, the following example would not work:

```
$ node -e 'process.env.foo = "bar"' && echo $foo
```

While the following will:

```
process.env.foo = 'bar';
console.log(process.env.foo);
```

Assigning a property on `process.env` will implicitly convert the value to a string.

Example:

```
process.env.test = null;
console.log(process.env.test);
// => 'null'
process.env.test = undefined;
console.log(process.env.test);
// => 'undefined'
```

Use `delete` to delete a property from `process.env`.

Example:

```
process.env.TEST = 1;
delete process.env.TEST;
console.log(process.env.TEST);
// => undefined
```

On Windows operating systems, environment variables are case-insensitive.

Example:

```
process.env.TEST = 1;
console.log(process.env.test);
// => 1
```

process.execArgv

Added in: v0.7.7

- <Array>

The `process.execArgv` property returns the set of Node.js-specific command-line options passed when the Node.js process was launched. These options do not appear in the array returned by the process.argv property, and do not include the Node.js executable, the name of the script, or any options following the script name. These options are useful in order to spawn child processes with the same execution environment as the parent.

```
$ node --harmony script.js --version
```

Results in `process.execArgv`:

```
['--harmony']
```

And `process.argv`:

```
['/usr/local/bin/node', 'script.js', '--version']
```

process.execPath

Added in: v0.1.100

- <string>

The `process.execPath` property returns the absolute pathname of the executable that started the Node.js process.

```
'/usr/local/bin/node'
```

process.exit([code])

Added in: v0.1.13

- `code` <integer> The exit code. Defaults to `0`.

The `process.exit()` method instructs Node.js to terminate the process synchronously with an exit status of `code`. If `code` is omitted, exit uses either the 'success' code `0` or the value of `process.exitCode` if it has been set. Node.js will not terminate until all the 'exit' event listeners are called.

To exit with a 'failure' code:

```
process.exit(1);
```

The shell that executed Node.js should see the exit code as `1`.

Calling `process.exit()` will force the process to exit as quickly as possible even if there are still asynchronous operations pending that have not yet completed fully, including I/O operations to `process.stdout` and `process.stderr`.

In most situations, it is not actually necessary to call `process.exit()` explicitly. The Node.js process will exit on its own *if there is no additional work pending* in the event loop. The `process.exitCode` property can be set to tell the process which exit code to use when the process exits gracefully.

For instance, the following example illustrates a *misuse* of the `process.exit()` method that could lead to data printed to stdout being truncated and lost:

```
// This is an example of what *not* to do:
if (someConditionNotMet()) {
  printUsageToStdout();
  process.exit(1);
}
```

The reason this is problematic is because writes to `process.stdout` in Node.js are sometimes *asynchronous* and may occur over multiple ticks of the Node.js event loop. Calling `process.exit()`, however, forces the process to exit *before* those additional writes to `stdout` can be performed.

Rather than calling `process.exit()` directly, the code *should* set the `process.exitCode` and allow the process to exit naturally by avoiding scheduling any additional work for the event loop:

```
// How to properly set the exit code while letting
// the process exit gracefully.
if (someConditionNotMet()) {
  printUsageToStdout();
  process.exitCode = 1;
}
```

If it is necessary to terminate the Node.js process due to an error condition, throwing an *uncaught* error and allowing the process to terminate accordingly is safer than calling `process.exit()`.

process.exitCode

Added in: v0.11.8

- <integer>

A number which will be the process exit code, when the process either exits gracefully, or is exited via process.exit() without specifying a code.

Specifying a code to process.exit(code) will override any previous setting of `process.exitCode`.

process.getegid()

Added in: v2.0.0

The `process.getegid()` method returns the numerical effective group identity of the Node.js process. (See getegid(2) .)

```
if (process.getegid) {
  console.log(`Current gid: ${process.getegid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.geteuid()

Added in: v2.0.0

- Returns: <Object>

The `process.geteuid()` method returns the numerical effective user identity of the process. (See geteuid(2) .)

```
if (process.geteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getgid()

Added in: v0.1.31

- Returns: <Object>

The `process.getgid()` method returns the numerical group identity of the process. (See getgid(2) .)

```
if (process.getgid) {
  console.log(`Current gid: ${process.getgid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getgroups()

Added in: v0.9.4

- Returns: <Array>

The `process.getgroups()` method returns an array with the supplementary group IDs. POSIX leaves it unspecified if the effective group ID is included but Node.js ensures it always is.

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getuid()

Added in: v0.1.28

- Returns: <integer>

The `process.getuid()` method returns the numeric user identity of the process. (See getuid(2) .)

```
if (process.getuid) {
  console.log(`Current uid: ${process.getuid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.hasUncaughtExceptionCaptureCallback()

Added in: v9.3.0

- Returns: <boolean>

Indicates whether a callback has been set using process.setUncaughtExceptionCaptureCallback() .

process.hrtime([time])

Added in: v0.7.6

- `time` <Array> The result of a previous call to `process.hrtime()`
- Returns: <Array>

The `process.hrtime()` method returns the current high-resolution real time in a `[seconds, nanoseconds]` tuple Array, where `nanoseconds` is the remaining part of the real time that can't be represented in second precision.

`time` is an optional parameter that must be the result of a previous `process.hrtime()` call to diff with the current time. If the parameter passed in is not a tuple Array, a `TypeError` will be thrown. Passing in a user-defined array instead of the result of a previous call to `process.hrtime()` will lead to undefined behavior.

These times are relative to an arbitrary time in the past, and not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals:

```
const NS_PER_SEC = 1e9;
const time = process.hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
  const diff = process.hrtime(time);
  // [ 1, 552 ]

  console.log(`Benchmark took ${diff[0] * NS_PER_SEC + diff[1]} nanoseconds`);
  // benchmark took 1000000552 nanoseconds
}, 1000);
```

process.initgroups(user, extra_group)

Added in: v0.9.4

- `user` <string> | <number> The user name or numeric identifier.
- `extra_group` <string> | <number> A group name or numeric identifier.

The `process.initgroups()` method reads the `/etc/group` file and initializes the group access list, using all groups of which the user is a member. This is a privileged operation that requires that the Node.js process either have `root` access or the `CAP_SETGID` capability.

Note that care must be taken when dropping privileges. Example:

```
console.log(process.getgroups());         // [ 0 ]
process.initgroups('bnoordhuis', 1000);   // switch user
console.log(process.getgroups());         // [ 27, 30, 46, 1000, 0 ]
process.setgid(1000);                     // drop root gid
console.log(process.getgroups());         // [ 27, 30, 46, 1000 ]
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.kill(pid[, signal])

Added in: v0.0.6

- `pid` <number> A process ID
- `signal` <string> | <number> The signal to send, either as a string or number. Defaults to `'SIGTERM'`.

The `process.kill()` method sends the `signal` to the process identified by `pid`.

Signal names are strings such as `'SIGINT'` or `'SIGHUP'`. See Signal Events and kill(2) for more information.

This method will throw an error if the target `pid` does not exist. As a special case, a signal of `0` can be used to test for the existence of a process. Windows platforms will throw an error if the `pid` is used to kill a process group.

Even though the name of this function is `process.kill()`, it is really just a signal sender, like the `kill` system call. The signal sent may do something other than kill the target process.

```
process.on('SIGHUP', () => {
  console.log('Got SIGHUP signal.');
});

setTimeout(() => {
  console.log('Exiting.');
  process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

When `SIGUSR1` is received by a Node.js process, Node.js will start the debugger, see Signal Events .

process.mainModule

Added in: v0.1.17

- <Object>

The `process.mainModule` property provides an alternative way of retrieving require.main . The difference is that if the main module changes at runtime, require.main may still refer to the original main module in modules that were required before the change occurred. Generally, it's safe to assume that the two refer to the same module.

As with require.main , `process.mainModule` will be `undefined` if there is no entry script.

process.memoryUsage()

▸ History

- Returns: <Object>
  - `rss` <integer>
  - `heapTotal` <integer>
  - `heapUsed` <integer>
  - `external` <integer>

The `process.memoryUsage()` method returns an object describing the memory usage of the Node.js process measured in bytes.

For example, the code:

```
console.log(process.memoryUsage());
```

Will generate:

```
{
  rss: 4935680,
  heapTotal: 1826816,
  heapUsed: 650472,
  external: 49879
}
```

`heapTotal` and `heapUsed` refer to V8's memory usage. `external` refers to the memory usage of C++ objects bound to JavaScript objects managed by V8. `rss` , Resident Set Size, is the amount of space occupied in the main memory device (that is a subset of the total allocated memory) for the process, which includes the *heap*, *code segment* and *stack*.

The *heap* is where objects, strings, and closures are stored. Variables are stored in the *stack* and the actual JavaScript code resides in the *code segment*.

process.nextTick(callback[, ...args])

▸ History

- `callback` <Function>
- `...args` <any> Additional arguments to pass when invoking the `callback`

The `process.nextTick()` method adds the `callback` to the "next tick queue". Once the current turn of the event loop turn runs to completion, all callbacks currently in the next tick queue will be called.

This is *not* a simple alias to `setTimeout(fn, 0)`. It is much more efficient. It runs before any additional I/O events (including timers) fire in subsequent ticks of the event loop.

```
console.log('start');
process.nextTick(() => {
  console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
```

This is important when developing APIs in order to give users the opportunity to assign event handlers *after* an object has been constructed but before any I/O has occurred:

```
function MyThing(options) {
  this.setupOptions(options);

  process.nextTick(() => {
    this.startDoingStuff();
  });
}

const thing = new MyThing();
thing.getReadyForStuff();

// thing.startDoingStuff() gets called now, not before.
```

It is very important for APIs to be either 100% synchronous or 100% asynchronous. Consider this example:

```
// WARNING!  DO NOT USE!  BAD UNSAFE HAZARD!
function maybeSync(arg, cb) {
  if (arg) {
    cb();
    return;
  }

  fs.stat('file', cb);
}
```

This API is hazardous because in the following case:

```
const maybeTrue = Math.random() > 0.5;

maybeSync(maybeTrue, () => {
  foo();
});

bar();
```

It is not clear whether `foo()` or `bar()` will be called first.

The following approach is much better:

```
function definitelyAsync(arg, cb) {
  if (arg) {
    process.nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}
```

The next tick queue is completely drained on each pass of the event loop **before** additional I/O is processed. As a result, recursively setting nextTick callbacks will block any I/O from happening, just like a `while(true);` loop.

process.noDeprecation

Added in: v0.8.0

- <boolean>

The `process.noDeprecation` property indicates whether the `--no-deprecation` flag is set on the current Node.js process. See the documentation for the `warning` event and the `emitWarning` method for more information about this flag's behavior.

process.pid

Added in: v0.1.15

- <integer>

The `process.pid` property returns the PID of the process.

```
console.log(`This process is pid ${process.pid}`);
```

process.platform

Added in: v0.1.16

- <string>

The `process.platform` property returns a string identifying the operating system platform on which the Node.js process is running.

Currently possible values are:

- `'aix'`
- `'darwin'`
- `'freebsd'`
- `'linux'`
- `'openbsd'`

- 'sunos'
- 'win32'

```
console.log(`This platform is ${process.platform}`);
```

The value 'android' may also be returned if the Node.js is built on the Android operating system. However, Android support in Node.js is experimental .

process.ppid

Added in: v9.2.0

- <integer>

The process.ppid property returns the PID of the current parent process.

```
console.log(`The parent process is pid ${process.ppid}`);
```

process.release

▸ History

- <Object>

The process.release property returns an Object containing metadata related to the current release, including URLs for the source tarball and headers-only tarball.

process.release contains the following properties:

- name <string> A value that will always be 'node' for Node.js. For legacy io.js releases, this will be 'io.js'.
- sourceUrl <string> an absolute URL pointing to a .tar.gz file containing the source code of the current release.
- headersUrl <string> an absolute URL pointing to a .tar.gz file containing only the source header files for the current release. This file is significantly smaller than the full source file and can be used for compiling Node.js native add-ons.
- libUrl <string> an absolute URL pointing to a node.lib file matching the architecture and version of the current release. This file is used for compiling Node.js native add-ons. *This property is only present on Windows builds of Node.js and will be missing on all other platforms.*
- lts <string> a string label identifying the LTS label for this release. This property only exists for LTS releases and is undefined for all other release types, including *Current* releases. Currently the valid values are:
  - 'Argon' for the 4.x LTS line beginning with 4.2.0.
  - 'Boron' for the 6.x LTS line beginning with 6.9.0.
  - 'Carbon' for the 8.x LTS line beginning with 8.9.1.

```
{
  name: 'node',
  lts: 'Argon',
  sourceUrl: 'https://nodejs.org/download/release/v4.4.5/node-v4.4.5.tar.gz',
  headersUrl: 'https://nodejs.org/download/release/v4.4.5/node-v4.4.5-headers.tar.gz',
  libUrl: 'https://nodejs.org/download/release/v4.4.5/win-x64/node.lib'
}
```

In custom builds from non-release versions of the source tree, only the name property may be present. The additional properties should not be relied upon to exist.

process.send(message[, sendHandle[, options]][, callback])

Added in: v0.5.9

- message <Object>

- `sendHandle` <Handle object>
- `options` <Object>
- `callback` <Function>
- Returns: <boolean>

If Node.js is spawned with an IPC channel, the `process.send()` method can be used to send messages to the parent process. Messages will be received as a `'message'` event on the parent's `ChildProcess` object.

If Node.js was not spawned with an IPC channel, `process.send()` will be `undefined`.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

process.setegid(id)

Added in: v2.0.0

- `id` <string> | <number> A group name or ID

The `process.setegid()` method sets the effective group identity of the process. (See setegid(2).) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated a numeric ID.

```
if (process.getegid && process.setegid) {
  console.log(`Current gid: ${process.getegid()}`);
  try {
    process.setegid(501);
    console.log(`New gid: ${process.getegid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.seteuid(id)

Added in: v2.0.0

- `id` <string> | <number> A user name or ID

The `process.seteuid()` method sets the effective user identity of the process. (See seteuid(2).) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

```
if (process.geteuid && process.seteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
  try {
    process.seteuid(501);
    console.log(`New uid: ${process.geteuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.setgid(id)

Added in: v0.1.31

- `id` <string> | <number> The group name or ID

The `process.setgid()` method sets the group identity of the process. (See setgid(2) .) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated numeric ID.

```
if (process.getgid && process.setgid) {
  console.log(`Current gid: ${process.getgid()}`);
  try {
    process.setgid(501);
    console.log(`New gid: ${process.getgid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.setgroups(groups)

Added in: v0.9.4

- `groups` <Array>

The `process.setgroups()` method sets the supplementary group IDs for the Node.js process. This is a privileged operation that requires the Node.js process to have `root` or the `CAP_SETGID` capability.

The `groups` array can contain numeric group IDs, group names or both.

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.setuid(id)

Added in: v0.1.28

The `process.setuid(id)` method sets the user identity of the process. (See setuid(2) .) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

```
if (process.getuid && process.setuid) {
  console.log(`Current uid: ${process.getuid()}`);
  try {
    process.setuid(501);
    console.log(`New uid: ${process.getuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.setUncaughtExceptionCaptureCallback(fn)

Added in: v9.3.0

- `fn` <Function> | <null>

The `process.setUncaughtExceptionCapture` function sets a function that will be invoked when an uncaught exception occurs, which will receive the exception value itself as its first argument.

If such a function is set, the `process.on('uncaughtException')` event will not be emitted. If `--abort-on-uncaught-exception` was passed from the command line or set through `v8.setFlagsFromString()` , the process will not abort.

To unset the capture function, `process.setUncaughtExceptionCapture(null)` may be used. Calling this method with a non-`null` argument while another capture function is set will throw an error.

Using this function is mutually exclusive with using the deprecated `domain` built-in module.

process.stderr

- <Stream>

The `process.stderr` property returns a stream connected to `stderr` (fd `2`). It is a `net.Socket` (which is a Duplex stream) unless fd `2` refers to a file, in which case it is a Writable stream.

`process.stderr` differs from other Node.js streams in important ways, see note on process I/O for more information.

process.stdin

- <Stream>

The `process.stdin` property returns a stream connected to `stdin` (fd `0`). It is a `net.Socket` (which is a Duplex stream) unless fd `0` refers to a file, in which case it is a Readable stream.

```
process.stdin.setEncoding('utf8');

process.stdin.on('readable', () => {
  const chunk = process.stdin.read();
  if (chunk !== null) {
    process.stdout.write(`data: ${chunk}`);
  }
});

process.stdin.on('end', () => {
  process.stdout.write('end');
});
```

As a Duplex stream, `process.stdin` can also be used in "old" mode that is compatible with scripts written for Node.js prior to v0.10. For more information see Stream compatibility.

In "old" streams mode the `stdin` stream is paused by default, so one must call `process.stdin.resume()` to read from it. Note also that calling `process.stdin.resume()` itself would switch stream to "old" mode.

process.stdout

- <Stream>

The `process.stdout` property returns a stream connected to `stdout` (fd `1`). It is a `net.Socket` (which is a Duplex stream) unless fd `1` refers to a file, in which case it is a Writable stream.

For example, to copy process.stdin to process.stdout:

```
process.stdin.pipe(process.stdout);
```

`process.stdout` differs from other Node.js streams in important ways, see note on process I/O for more information.

A note on process I/O

`process.stdout` and `process.stderr` differ from other Node.js streams in important ways:

1. They are used internally by `console.log()` and `console.error()`, respectively.
2. They cannot be closed (`end()` will throw).
3. They will never emit the `'finish'` event.
4. Writes may be synchronous depending on what the stream is connected to and whether the system is Windows or POSIX:

- Files: *synchronous* on Windows and POSIX
- TTYs (Terminals): *asynchronous* on Windows, *synchronous* on POSIX
- Pipes (and sockets): *synchronous* on Windows, *asynchronous* on POSIX

These behaviors are partly for historical reasons, as changing them would create backwards incompatibility, but they are also expected by some users.

Synchronous writes avoid problems such as output written with `console.log()` or `console.error()` being unexpectedly interleaved, or not written at all if `process.exit()` is called before an asynchronous write completes. See process.exit() for more information.

Warning: Synchronous writes block the event loop until the write has completed. This can be near instantaneous in the case of output to a file, but under high system load, pipes that are not being read at the receiving end, or with slow terminals or file systems, its possible for the event loop to be blocked often enough and long enough to have severe negative performance impacts. This may not be a problem when writing to an interactive terminal session, but consider this particularly careful when doing production logging to the process output streams.

To check if a stream is connected to a TTY context, check the `isTTY` property.

For instance:

```
$ node -p "Boolean(process.stdin.isTTY)"
true
$ echo "foo" | node -p "Boolean(process.stdin.isTTY)"
false
$ node -p "Boolean(process.stdout.isTTY)"
true
$ node -p "Boolean(process.stdout.isTTY)" | cat
false
```

See the TTY documentation for more information.

process.throwDeprecation

Added in: v0.9.12

- <boolean>

The `process.throwDeprecation` property indicates whether the `--throw-deprecation` flag is set on the current Node.js process. See the documentation for the warning event and the emitWarning method for more information about this flag's behavior.

process.title

Added in: v0.1.104

- <string>

The `process.title` property returns the current process title (i.e. returns the current value of `ps`). Assigning a new value to `process.title` modifies the current value of `ps`.

When a new value is assigned, different platforms will impose different maximum length restrictions on the title. Usually such restrictions are quite limited. For instance, on Linux and macOS, `process.title` is limited to the size of the binary name plus the length of the command line arguments because setting the `process.title` overwrites the `argv` memory of the process. Node.js v0.8 allowed for longer process title strings by also overwriting the `environ` memory but that was potentially insecure and confusing in some (rather obscure) cases.

process.traceDeprecation

Added in: v0.8.0

- <boolean>

The `process.traceDeprecation` property indicates whether the `--trace-deprecation` flag is set on the current Node.js process. See the documentation for the warning event and the emitWarning method for more information about this flag's behavior.

process.umask([mask])

Added in: v0.1.19

- `mask` <number>

The `process.umask()` method sets or returns the Node.js process's file mode creation mask. Child processes inherit the mask from the parent process. Invoked without an argument, the current mask is returned, otherwise the umask is set to the argument value and the previous mask is returned.

```
const newmask = 0o022;
const oldmask = process.umask(newmask);
console.log(
  `Changed umask from ${oldmask.toString(8)} to ${newmask.toString(8)}`
);
```

process.uptime()

Added in: v0.5.0

- Returns: <number>

The `process.uptime()` method returns the number of seconds the current Node.js process has been running.

The return value includes fractions of a second. Use `Math.floor()` to get whole seconds.

process.version

Added in: v0.1.3

- <string>

The `process.version` property returns the Node.js version string.

```
console.log(`Version: ${process.version}`);
```

process.versions

▸ History

- <Object>

The `process.versions` property returns an object listing the version strings of Node.js and its dependencies. `process.versions.modules` indicates the current ABI version, which is increased whenever a C++ API changes. Node.js will refuse to load modules that were compiled against a different module ABI version.

```
console.log(process.versions);
```

Will generate an object similar to:

```
{ http_parser: '2.7.0',
  node: '8.9.0',
  v8: '6.3.292.48-node.6',
  uv: '1.18.0',
  zlib: '1.2.11',
  ares: '1.13.0',
  modules: '60',
  nghttp2: '1.29.0',
  napi: '2',
  openssl: '1.0.2n',
  icu: '60.1',
  unicode: '10.0',
  cldr: '32.0',
  tz: '2016b' }
```

Exit Codes

Node.js will normally exit with a `0` status code when no more async operations are pending. The following status codes are used in other cases:

- `1` **Uncaught Fatal Exception** - There was an uncaught exception, and it was not handled by a domain or an `'uncaughtException'` event handler.
- `2` - Unused (reserved by Bash for builtin misuse)
- `3` **Internal JavaScript Parse Error** - The JavaScript source code internal in Node.js's bootstrapping process caused a parse error. This is extremely rare, and generally can only happen during development of Node.js itself.
- `4` **Internal JavaScript Evaluation Failure** - The JavaScript source code internal in Node.js's bootstrapping process failed to return a function value when evaluated. This is extremely rare, and generally can only happen during development of Node.js itself.
- `5` **Fatal Error** - There was a fatal unrecoverable error in V8. Typically a message will be printed to stderr with the prefix `FATAL ERROR`.
- `6` **Non-function Internal Exception Handler** - There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called.
- `7` **Internal Exception Handler Run-Time Failure** - There was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it. This can happen, for example, if a `'uncaughtException'` or `domain.on('error')` handler throws an error.
- `8` - Unused. In previous versions of Node.js, exit code 8 sometimes indicated an uncaught exception.
- `9` - **Invalid Argument** - Either an unknown option was specified, or an option requiring a value was provided without a value.
- `10` **Internal JavaScript Run-Time Failure** - The JavaScript source code internal in Node.js's bootstrapping process threw an error when the bootstrapping function was called. This is extremely rare, and generally can only happen during development of Node.js itself.
- `12` **Invalid Debug Argument** - The `--inspect` and/or `--inspect-brk` options were set, but the port number chosen was invalid or unavailable.
- `>128` **Signal Exits** - If Node.js receives a fatal signal such as `SIGKILL` or `SIGHUP`, then its exit code will be `128` plus the value of the signal code. This is a standard POSIX practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code. For example, signal `SIGABRT` has value `6`, so the expected exit code will be `128` + `6`, or `134`.

Punycode

▸ History

Stability: 0 - Deprecated

The version of the punycode module bundled in Node.js is being deprecated. In a future major version of Node.js this module will be removed. Users currently depending on the `punycode` module should switch to using the userland-provided Punycode.js module instead.

The `punycode` module is a bundled version of the Punycode.js module. It can be accessed using:

```
const punycode = require('punycode');
```

Punycode is a character encoding scheme defined by RFC 3492 that is primarily intended for use in Internationalized Domain Names. Because host names in URLs are limited to ASCII characters only, Domain Names that contain non-ASCII characters must be converted into ASCII using the Punycode scheme. For instance, the Japanese character that translates into the English word, `'example'` is `'□'`. The Internationalized Domain Name, `'□.com'` (equivalent to `'example.com'`) is represented by Punycode as the ASCII string `'xn--fsq.com'`.

The `punycode` module provides a simple implementation of the Punycode standard.

The `punycode` module is a third-party dependency used by Node.js and made available to developers as a convenience. Fixes or other modifications to the module must be directed to the Punycode.js project.

punycode.decode(string)

Added in: v0.5.1

- `string` <string>

The `punycode.decode()` method converts a Punycode string of ASCII-only characters to the equivalent string of Unicode codepoints.

```
punycode.decode('maana-pta'); // 'mañana'
punycode.decode('--dqo34k'); // '☃-⌘'
```

punycode.encode(string)

Added in: v0.5.1

- `string` <string>

The `punycode.encode()` method converts a string of Unicode codepoints to a Punycode string of ASCII-only characters.

```
punycode.encode('mañana'); // 'maana-pta'
punycode.encode('☃-⌘'); // '--dqo34k'
```

punycode.toASCII(domain)

Added in: v0.6.1

- `domain` <string>

The `punycode.toASCII()` method converts a Unicode string representing an Internationalized Domain Name to Punycode. Only the non-ASCII parts of the domain name will be converted. Calling `punycode.toASCII()` on a string that already only contains ASCII characters will have no effect.

```
// encode domain names
punycode.toASCII('mañana.com');  // 'xn--maana-pta.com'
punycode.toASCII('☃-⌘.com');   // 'xn----dqo34k.com'
punycode.toASCII('example.com'); // 'example.com'
```

punycode.toUnicode(domain)

Added in: v0.6.1

- `domain` <string>

The `punycode.toUnicode()` method converts a string representing a domain name containing Punycode encoded characters into Unicode. Only the Punycode encoded parts of the domain name are be converted.

```
// decode domain names
punycode.toUnicode('xn--maana-pta.com'); // 'mañana.com'
punycode.toUnicode('xn----dqo34k.com');  // '☃-⌘.com'
punycode.toUnicode('example.com');       // 'example.com'
```

punycode.ucs2

Added in: v0.7.0

punycode.ucs2.decode(string)

Added in: v0.7.0

- `string` <string>

The `punycode.ucs2.decode()` method returns an array containing the numeric codepoint values of each Unicode symbol in the string.

```
punycode.ucs2.decode('abc'); // [0x61, 0x62, 0x63]
// surrogate pair for U+1D306 tetragram for centre:
punycode.ucs2.decode('\uD834\uDF06'); // [0x1D306]
```

punycode.ucs2.encode(codePoints)

Added in: v0.7.0

- `codePoints` <Array>

The `punycode.ucs2.encode()` method returns a string based on an array of numeric code point values.

```
punycode.ucs2.encode([0x61, 0x62, 0x63]); // 'abc'
punycode.ucs2.encode([0x1D306]); // '\uD834\uDF06'
```

punycode.version

Added in: v0.6.1

Returns a string identifying the current Punycode.js version number.

Query String

Stability: 2 - Stable

The `querystring` module provides utilities for parsing and formatting URL query strings. It can be accessed using:

```
const querystring = require('querystring');
```

querystring.escape(str)

Added in: v0.1.25

- `str` <string>

The `querystring.escape()` method performs URL percent-encoding on the given `str` in a manner that is optimized for the specific requirements of URL query strings.

The `querystring.escape()` method is used by `querystring.stringify()` and is generally not expected to be used directly. It is exported primarily to

allow application code to provide a replacement percent-encoding implementation if necessary by assigning `querystring.escape` to an alternative function.

querystring.parse(str[, sep[, eq[, options]]])

▸ History

- `str` <string> The URL query string to parse
- `sep` <string> The substring used to delimit key and value pairs in the query string. Defaults to `'&'`.
- `eq` <string>. The substring used to delimit keys and values in the query string. Defaults to `'='`.
- `options` <Object>
  - `decodeURIComponent` <Function> The function to use when decoding percent-encoded characters in the query string. Defaults to `querystring.unescape()`.
  - `maxKeys` <number> Specifies the maximum number of keys to parse. Defaults to `1000`. Specify `0` to remove key counting limitations.

The `querystring.parse()` method parses a URL query string (`str`) into a collection of key and value pairs.

For example, the query string `'foo=bar&abc=xyz&abc=123'` is parsed into:

```
{
  foo: 'bar',
  abc: ['xyz', '123']
}
```

The object returned by the `querystring.parse()` method *does not* prototypically inherit from the JavaScript `Object`. This means that typical `Object` methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and *will not work*.

By default, percent-encoded characters within the query string will be assumed to use UTF-8 encoding. If an alternative character encoding is used, then an alternative `decodeURIComponent` option will need to be specified as illustrated in the following example:

```
// Assuming gbkDecodeURIComponent function already exists...

querystring.parse('w=%D6%D0%CE%C4&foo=bar', null, null,
                  { decodeURIComponent: gbkDecodeURIComponent });
```

querystring.stringify(obj[, sep[, eq[, options]]])

Added in: v0.1.25

- `obj` <Object> The object to serialize into a URL query string
- `sep` <string> The substring used to delimit key and value pairs in the query string. Defaults to `'&'`.
- `eq` <string>. The substring used to delimit keys and values in the query string. Defaults to `'='`.
- `options`
  - `encodeURIComponent` <Function> The function to use when converting URL-unsafe characters to percent-encoding in the query string. Defaults to `querystring.escape()`.

The `querystring.stringify()` method produces a URL query string from a given `obj` by iterating through the object's "own properties".

It serializes the following types of values passed in `obj`: <string> | <number> | <boolean> | <string[]> | <number[]> | <boolean[]> Any other input values will be coerced to empty strings.

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' });
// returns 'foo=bar&baz=qux&baz=quux&corge='

querystring.stringify({ foo: 'bar', baz: 'qux' }, ';', ':');
// returns 'foo:bar;baz:qux'
```

By default, characters requiring percent-encoding within the query string will be encoded as UTF-8. If an alternative encoding is required, then an alternative `encodeURIComponent` option will need to be specified as illustrated in the following example:

```
// Assuming gbkEncodeURIComponent function already exists,

querystring.stringify({ w: '□□', foo: 'bar' }, null, null,
                      { encodeURIComponent: gbkEncodeURIComponent });
```

querystring.unescape(str)

Added in: v0.1.25

- `str` <string>

The `querystring.unescape()` method performs decoding of URL percent-encoded characters on the given `str`.

The `querystring.unescape()` method is used by `querystring.parse()` and is generally not expected to be used directly. It is exported primarily to allow application code to provide a replacement decoding implementation if necessary by assigning `querystring.unescape` to an alternative function.

By default, the `querystring.unescape()` method will attempt to use the JavaScript built-in `decodeURIComponent()` method to decode. If that fails, a safer equivalent that does not throw on malformed URLs will be used.

Readline

Stability: 2 - Stable

The `readline` module provides an interface for reading data from a Readable stream (such as process.stdin) one line at a time. It can be accessed using:

```
const readline = require('readline');
```

The following simple example illustrates the basic use of the `readline` module.

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('What do you think of Node.js? ', (answer) => {
  // TODO: Log the answer in a database
  console.log(`Thank you for your valuable feedback: ${answer}`);

  rl.close();
});
```

Once this code is invoked, the Node.js application will not terminate until the `readline.Interface` is closed because the interface waits for data

to be received on the `input` stream.

Class: Interface

Added in: v0.1.104

Instances of the `readline.Interface` class are constructed using the `readline.createInterface()` method. Every instance is associated with a single `input` Readable stream and a single `output` Writable stream. The `output` stream is used to print prompts for user input that arrives on, and is read from, the `input` stream.

Event: 'close'

Added in: v0.1.98

The `'close'` event is emitted when one of the following occur:

- The `rl.close()` method is called and the `readline.Interface` instance has relinquished control over the `input` and `output` streams;
- The `input` stream receives its `'end'` event;
- The `input` stream receives `<ctrl>-D` to signal end-of-transmission (EOT);
- The `input` stream receives `<ctrl>-C` to signal `SIGINT` and there is no `SIGINT` event listener registered on the `readline.Interface` instance.

The listener function is called without passing any arguments.

The `readline.Interface` instance is finished once the `'close'` event is emitted.

Event: 'line'

Added in: v0.1.98

The `'line'` event is emitted whenever the `input` stream receives an end-of-line input (`\n`, `\r`, or `\r\n`). This usually occurs when the user presses the `<Enter>`, or `<Return>` keys.

The listener function is called with a string containing the single line of received input.

```
rl.on('line', (input) => {
  console.log(`Received: ${input}`);
});
```

Event: 'pause'

Added in: v0.7.5

The `'pause'` event is emitted when one of the following occur:

- The `input` stream is paused.
- The `input` stream is not paused and receives the `SIGCONT` event. (See events `SIGTSTP` and `SIGCONT`)

The listener function is called without passing any arguments.

```
rl.on('pause', () => {
  console.log('Readline paused.');
});
```

Event: 'resume'

Added in: v0.7.5

The `'resume'` event is emitted whenever the `input` stream is resumed.

The listener function is called without passing any arguments.

```
rl.on('resume', () => {
  console.log('Readline resumed.');
});
```

Event: 'SIGCONT'

Added in: v0.7.5

The `'SIGCONT'` event is emitted when a Node.js process previously moved into the background using `<ctrl>-Z` (i.e. `SIGTSTP`) is then brought back to the foreground using fg(1p).

If the `input` stream was paused *before* the `SIGTSTP` request, this event will not be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGCONT', () => {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```

The `'SIGCONT'` event is *not* supported on Windows.

Event: 'SIGINT'

Added in: v0.3.0

The `'SIGINT'` event is emitted whenever the `input` stream receives a `<ctrl>-C` input, known typically as `SIGINT`. If there are no `'SIGINT'` event listeners registered when the `input` stream receives a `SIGINT`, the `'pause'` event will be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGINT', () => {
  rl.question('Are you sure you want to exit? ', (answer) => {
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```

Event: 'SIGTSTP'

Added in: v0.7.5

The `'SIGTSTP'` event is emitted when the `input` stream receives a `<ctrl>-Z` input, typically known as `SIGTSTP`. If there are no `SIGTSTP` event listeners registered when the `input` stream receives a `SIGTSTP`, the Node.js process will be sent to the background.

When the program is resumed using fg(1p), the `'pause'` and `SIGCONT` events will be emitted. These can be used to resume the `input` stream.

The `'pause'` and `'SIGCONT'` events will not be emitted if the `input` was paused before the process was sent to the background.

The listener function is invoked without passing any arguments.

```
rl.on('SIGTSTP', () => {
  // This will override SIGTSTP and prevent the program from going to the
  // background.
  console.log('Caught SIGTSTP.');
});
```

The `'SIGTSTP'` event is *not* supported on Windows.

rl.close()

Added in: v0.1.98

The `rl.close()` method closes the `readline.Interface` instance and relinquishes control over the `input` and `output` streams. When called, the `'close'` event will be emitted.

rl.pause()

Added in: v0.3.4

The `rl.pause()` method pauses the `input` stream, allowing it to be resumed later if necessary.

Calling `rl.pause()` does not immediately pause other events (including `'line'`) from being emitted by the `readline.Interface` instance.

rl.prompt([preserveCursor])

Added in: v0.1.98

- `preserveCursor` <boolean> If `true`, prevents the cursor placement from being reset to `0`.

The `rl.prompt()` method writes the `readline.Interface` instances configured `prompt` to a new line in `output` in order to provide a user with a new location at which to provide input.

When called, `rl.prompt()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the prompt is not written.

rl.question(query, callback)

Added in: v0.3.3

- `query` <string> A statement or query to write to `output`, prepended to the prompt.
- `callback` <Function> A callback function that is invoked with the user's input in response to the `query`.

The `rl.question()` method displays the `query` by writing it to the `output`, waits for user input to be provided on `input`, then invokes the `callback` function passing the provided input as the first argument.

When called, `rl.question()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the `query` is not written.

Example usage:

```
rl.question('What is your favorite food? ', (answer) => {
  console.log(`Oh, so your favorite food is ${answer}`);
});
```

The `callback` function passed to `rl.question()` does not follow the typical pattern of accepting an `Error` object or `null` as the first argument. The `callback` is called with the provided answer as the only argument.

rl.resume()

Added in: v0.3.4

The `rl.resume()` method resumes the `input` stream if it has been paused.

rl.setPrompt(prompt)

Added in: v0.1.98

- `prompt` <string>

The `rl.setPrompt()` method sets the prompt that will be written to `output` whenever `rl.prompt()` is called.

rl.write(data[, key])

Added in: v0.1.98

- `data` <string>
- `key` <Object>
  - `ctrl` <boolean> `true` to indicate the `<ctrl>` key.
  - `meta` <boolean> `true` to indicate the `<Meta>` key.
  - `shift` <boolean> `true` to indicate the `<Shift>` key.
  - `name` <string> The name of the a key.

The `rl.write()` method will write either `data` or a key sequence identified by `key` to the `output`. The `key` argument is supported only if `output` is a TTY text terminal.

If `key` is specified, `data` is ignored.

When called, `rl.write()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the `data` and `key` are not written.

```
rl.write('Delete this!');
// Simulate Ctrl+u to delete the line written previously
rl.write(null, { ctrl: true, name: 'u' });
```

The `rl.write()` method will write the data to the `readline` Interface's `input` *as if it were provided by the user*.

readline.clearLine(stream, dir)

Added in: v0.7.7

- `stream` <stream.Writable>
- `dir` <number>
  - `-1` - to the left from cursor
  - `1` - to the right from cursor
  - `0` - the entire line

The `readline.clearLine()` method clears current line of given TTY stream in a specified direction identified by `dir`.

readline.clearScreenDown(stream)

Added in: v0.7.7

- `stream` <stream.Writable>

The `readline.clearScreenDown()` method clears the given TTY stream from the current position of the cursor down.

readline.createInterface(options)

▸ History

- `options` <Object>
  - `input` <stream.Readable> The Readable stream to listen to. This option is *required*.
  - `output` <stream.Writable> The Writable stream to write readline data to.
  - `completer` <Function> An optional function used for Tab autocompletion.
  - `terminal` <boolean> `true` if the `input` and `output` streams should be treated like a TTY, and have ANSI/VT100 escape codes written to it. Defaults to checking `isTTY` on the `output` stream upon instantiation.
  - `historySize` <number> Maximum number of history lines retained. To disable the history set this value to `0`. This option makes sense only if `terminal` is set to `true` by the user or by an internal `output` check, otherwise the history caching mechanism is not initialized at all. **Default:** `30`

- `prompt` <string> The prompt string to use. **Default:** `'> '`
- `crlfDelay` <number> If the delay between `\r` and `\n` exceeds `crlfDelay` milliseconds, both `\r` and `\n` will be treated as separate end-of-line input. `crlfDelay` will be coerced to a number no less than `100`. It can be set to `Infinity`, in which case `\r` followed by `\n` will always be considered a single newline (which may be reasonable for reading files with `\r\n` line delimiter). **Default:** `100`
- `removeHistoryDuplicates` <boolean> If `true`, when a new input line added to the history list duplicates an older one, this removes the older line from the list. **Default:** `false`

The `readline.createInterface()` method creates a new `readline.Interface` instance.

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

Once the `readline.Interface` instance is created, the most common case is to listen for the `'line'` event:

```
rl.on('line', (line) => {
  console.log(`Received: ${line}`);
});
```

If `terminal` is `true` for this instance then the `output` stream will get the best compatibility if it defines an `output.columns` property and emits a `'resize'` event on the `output` if or when the columns ever change (process.stdout does this automatically when it is a TTY).

Use of the `completer` Function

The `completer` function takes the current line entered by the user as an argument, and returns an Array with 2 entries:

- An Array with matching entries for the completion.
- The substring that was used for the matching.

For instance: `[[substr1, substr2, ...], originalsubstring]`.

```
function completer(line) {
  const completions = '.help .error .exit .quit .q'.split(' ');
  const hits = completions.filter((c) => c.startsWith(line));
  // show all completions if none found
  return [hits.length ? hits : completions, line];
}
```

The `completer` function can be called asynchronously if it accepts two arguments:

```
function completer(linePartial, callback) {
  callback(null, [['123'], linePartial]);
}
```

readline.cursorTo(stream, x, y)

Added in: v0.7.7

- `stream` <stream.Writable>
- `x` <number>
- `y` <number>

The `readline.cursorTo()` method moves cursor to the specified position in a given TTY `stream`.

readline.emitKeypressEvents(stream[, interface])

Added in: v0.7.7

- `stream` <stream.Readable>
- `interface` <readline.Interface>

The `readline.emitKeypressEvents()` method causes the given Readable `stream` to begin emitting `'keypress'` events corresponding to received input.

Optionally, `interface` specifies a `readline.Interface` instance for which autocompletion is disabled when copy-pasted input is detected.

If the `stream` is a TTY, then it must be in raw mode.

This is automatically called by any readline instance on its `input` if the `input` is a terminal. Closing the `readline` instance does not stop the `input` from emitting `'keypress'` events.

```
readline.emitKeypressEvents(process.stdin);
if (process.stdin.isTTY)
  process.stdin.setRawMode(true);
```

readline.moveCursor(stream, dx, dy)

Added in: v0.7.7

- `stream` <stream.Writable>
- `dx` <number>
- `dy` <number>

The `readline.moveCursor()` method moves the cursor *relative* to its current position in a given TTY `stream`.

Example: Tiny CLI

The following example illustrates the use of `readline.Interface` class to implement a small command-line interface:

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
  prompt: 'OHAI> '
});

rl.prompt();

rl.on('line', (line) => {
  switch (line.trim()) {
    case 'hello':
      console.log('world!');
      break;
    default:
      console.log(`Say what? I might have heard '${line.trim()}'`);
      break;
  }
  rl.prompt();
}).on('close', () => {
  console.log('Have a great day!');
  process.exit(0);
});
```

Example: Read File Stream Line-by-Line

A common use case for readline is to consume input from a filesystem Readable stream one line at a time, as illustrated in the following example:

```
const readline = require('readline');
const fs = require('fs');

const rl = readline.createInterface({
  input: fs.createReadStream('sample.txt'),
  crlfDelay: Infinity
});

rl.on('line', (line) => {
  console.log(`Line from file: ${line}`);
});
```

REPL

Stability: 2 - Stable

The repl module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications. It can be accessed using:

```
const repl = require('repl');
```

Design and Features

The repl module exports the repl.REPLServer class. While running, instances of repl.REPLServer will accept individual lines of user input, evaluate those according to a user-defined evaluation function, then output the result. Input and output may be from stdin and stdout, respectively, or may be connected to any Node.js stream.

Instances of repl.REPLServer support automatic completion of inputs, simplistic Emacs-style line editing, multi-line inputs, ANSI-styled output, saving and restoring current REPL session state, error recovery, and customizable evaluation functions.

Commands and Special Keys

The following special commands are supported by all REPL instances:

- .break - When in the process of inputting a multi-line expression, entering the .break command (or pressing the <ctrl>-C key combination) will abort further input or processing of that expression.
- .clear - Resets the REPL context to an empty object and clears any multi-line expression currently being input.
- .exit - Close the I/O stream, causing the REPL to exit.
- .help - Show this list of special commands.
- .save - Save the current REPL session to a file: > .save ./file/to/save.js
- .load - Load a file into the current REPL session. > .load ./file/to/load.js
- .editor - Enter editor mode (<ctrl>-D to finish, <ctrl>-C to cancel)

```
> .editor
// Entering editor mode (^D to finish, ^C to cancel)
function welcome(name) {
  return `Hello ${name}!`;
}

welcome('Node.js User');

// ^D
'Hello Node.js User!'
>
```

The following key combinations in the REPL have these special effects:

- `<ctrl>-C` - When pressed once, has the same effect as the `.break` command. When pressed twice on a blank line, has the same effect as the `.exit` command.
- `<ctrl>-D` - Has the same effect as the `.exit` command.
- `<tab>` - When pressed on a blank line, displays global and local(scope) variables. When pressed while entering other input, displays relevant autocompletion options.

Default Evaluation

By default, all instances of `repl.REPLServer` use an evaluation function that evaluates JavaScript expressions and provides access to Node.js' built-in modules. This default behavior can be overridden by passing in an alternative evaluation function when the `repl.REPLServer` instance is created.

JavaScript Expressions

The default evaluator supports direct evaluation of JavaScript expressions:

```
> 1 + 1
2
> const m = 2
undefined
> m + 1
3
```

Unless otherwise scoped within blocks or functions, variables declared either implicitly or using the `const`, `let`, or `var` keywords are declared at the global scope.

Global and Local Scope

The default evaluator provides access to any variables that exist in the global scope. It is possible to expose a variable to the REPL explicitly by assigning it to the `context` object associated with each `REPLServer`:

```
const repl = require('repl');
const msg = 'message';

repl.start('> ').context.m = msg;
```

Properties in the `context` object appear as local within the REPL:

```
$ node repl_test.js
> m
'message'
```

Context properties are not read-only by default. To specify read-only globals, context properties must be defined using `Object.defineProperty()`:

```
const repl = require('repl');
const msg = 'message';

const r = repl.start('> ');
Object.defineProperty(r.context, 'm', {
  configurable: false,
  enumerable: true,
  value: msg
});
```

Accessing Core Node.js Modules

The default evaluator will automatically load Node.js core modules into the REPL environment when used. For instance, unless otherwise declared as a global or scoped variable, the input `fs` will be evaluated on-demand as `global.fs = require('fs')`.

```
> fs.createReadStream('./some/file');
```

Assignment of the `_` (underscore) variable

▶ History

The default evaluator will, by default, assign the result of the most recently evaluated expression to the special variable `_` (underscore). Explicitly setting `_` to a value will disable this behavior.

```
> [ 'a', 'b', 'c' ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
Expression assignment to _ now disabled.
4
> 1 + 1
2
> _
4
```

Similarly, `_error` will refer to the last seen error, if there was any. Explicitly setting `_error` to a value will disable this behavior.

```
> throw new Error('foo');
Error: foo
> _error.message
'foo'
```

Custom Evaluation Functions

When a new `repl.REPLServer` is created, a custom evaluation function may be provided. This can be used, for instance, to implement fully customized REPL applications.

The following illustrates a hypothetical example of a REPL that performs translation of text from one language to another:

```
const repl = require('repl');
const { Translator } = require('translator');

const myTranslator = new Translator('en', 'fr');

function myEval(cmd, context, filename, callback) {
  callback(null, myTranslator.translate(cmd));
}

repl.start({ prompt: '> ', eval: myEval });
```

Recoverable Errors

As a user is typing input into the REPL prompt, pressing the `<enter>` key will send the current line of input to the `eval` function. In order to support multi-line input, the eval function can return an instance of `repl.Recoverable` to the provided callback function:

```
function myEval(cmd, context, filename, callback) {
  let result;
  try {
    result = vm.runInThisContext(cmd);
  } catch (e) {
    if (isRecoverableError(e)) {
      return callback(new repl.Recoverable(e));
    }
  }
  callback(null, result);
}

function isRecoverableError(error) {
  if (error.name === 'SyntaxError') {
    return /^(Unexpected end of input|Unexpected token)/.test(error.message);
  }
  return false;
}
```

Customizing REPL Output

By default, `repl.REPLServer` instances format output using the util.inspect() method before writing the output to the provided Writable stream (`process.stdout` by default). The `useColors` boolean option can be specified at construction to instruct the default writer to use ANSI style codes to colorize the output from the `util.inspect()` method.

It is possible to fully customize the output of a `repl.REPLServer` instance by passing a new function in using the `writer` option on construction. The following example, for instance, simply converts any input text to upper case:

```
const repl = require('repl');

const r = repl.start({ prompt: '> ', eval: myEval, writer: myWriter });

function myEval(cmd, context, filename, callback) {
  callback(null, cmd);
}

function myWriter(output) {
  return output.toUpperCase();
}
```

Class: REPLServer

Added in: v0.1.91

The `repl.REPLServer` class inherits from the readline.Interface class. Instances of `repl.REPLServer` are created using the `repl.start()` method and *should not* be created directly using the JavaScript `new` keyword.

Event: 'exit'

Added in: v0.7.7

The `'exit'` event is emitted when the REPL is exited either by receiving the `.exit` command as input, the user pressing `<ctrl>-C` twice to signal `SIGINT`, or by pressing `<ctrl>-D` to signal `'end'` on the input stream. The listener callback is invoked without any arguments.

```
replServer.on('exit', () => {
  console.log('Received "exit" event from repl!');
  process.exit();
});
```

Event: 'reset'

Added in: v0.11.0

The `'reset'` event is emitted when the REPL's context is reset. This occurs whenever the `.clear` command is received as input *unless* the REPL is using the default evaluator and the `repl.REPLServer` instance was created with the `useGlobal` option set to `true`. The listener callback will be called with a reference to the `context` object as the only argument.

This can be used primarily to re-initialize REPL context to some pre-defined state as illustrated in the following simple example:

```
const repl = require('repl');

function initializeContext(context) {
  context.m = 'test';
}

const r = repl.start({ prompt: '> ' });
initializeContext(r.context);

r.on('reset', initializeContext);
```

When this code is executed, the global `'m'` variable can be modified but then reset to its initial value using the `.clear` command:

```
$ ./node example.js
> m
'test'
> m = 1
1
> m
1
> .clear
Clearing context...
> m
'test'
>
```

replServer.defineCommand(keyword, cmd)

Added in: v0.3.0

- `keyword` <string> The command keyword (*without* a leading `.` character).
- `cmd` <Object> | <Function> The function to invoke when the command is processed.

The `replServer.defineCommand()` method is used to add new `.`-prefixed commands to the REPL instance. Such commands are invoked by typing a `.` followed by the `keyword`. The `cmd` is either a Function or an object with the following properties:

- `help` <string> Help text to be displayed when `.help` is entered (Optional).
- `action` <Function> The function to execute, optionally accepting a single string argument.

The following example shows two new commands added to the REPL instance:

```
const repl = require('repl');

const replServer = repl.start({ prompt: '> ' });
replServer.defineCommand('sayhello', {
  help: 'Say hello',
  action(name) {
    this.clearBufferedCommand();
    console.log(`Hello, ${name}!`);
    this.displayPrompt();
  }
});
replServer.defineCommand('saybye', function saybye() {
  console.log('Goodbye!');
  this.close();
});
```

The new commands can then be used from within the REPL instance:

```
> .sayhello Node.js User
Hello, Node.js User!
> .saybye
Goodbye!
```

replServer.displayPrompt([preserveCursor])

Added in: v0.1.91

- `preserveCursor` <boolean>

The `replServer.displayPrompt()` method readies the REPL instance for input from the user, printing the configured `prompt` to a new line in the `output` and resuming the `input` to accept new input.

When multi-line input is being entered, an ellipsis is printed rather than the 'prompt'.

When `preserveCursor` is `true`, the cursor placement will not be reset to `0`.

The `replServer.displayPrompt` method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

replServer.clearBufferedCommand()

Added in: v9.0.0

The `replServer.clearBufferedCommand()` method clears any command that has been buffered but not yet executed. This method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

replServer.parseREPLKeyword(keyword, [rest])

Added in: v0.8.9 Deprecated since: v9.0.0

- `keyword` <string> the potential keyword to parse and execute
- `rest` <any> any parameters to the keyword command

Stability: 0 - Deprecated.

An internal method used to parse and execute `REPLServer` keywords. Returns `true` if `keyword` is a valid keyword, otherwise `false`.

repl.start(options)

▶ History

- `options` <Object> | <string>
  - `prompt` <string> The input prompt to display. Defaults to `>` (with a trailing space).
  - `input` <stream.Readable> The Readable stream from which REPL input will be read. Defaults to `process.stdin`.
  - `output` <stream.Writable> The Writable stream to which REPL output will be written. Defaults to `process.stdout`.
  - `terminal` <boolean> If `true`, specifies that the `output` should be treated as a TTY terminal, and have ANSI/VT100 escape codes written to it. Defaults to checking the value of the `isTTY` property on the `output` stream upon instantiation.
  - `eval` <Function> The function to be used when evaluating each given line of input. **Default:** an async wrapper for the JavaScript `eval()` function. An `eval` function can error with `repl.Recoverable` to indicate the input was incomplete and prompt for additional lines.
  - `useColors` <boolean> If `true`, specifies that the default `writer` function should include ANSI color styling to REPL output. If a custom `writer` function is provided then this has no effect. Defaults to the REPL instances `terminal` value.
  - `useGlobal` <boolean> If `true`, specifies that the default evaluation function will use the JavaScript `global` as the context as opposed to creating a new separate context for the REPL instance. The node CLI REPL sets this value to `true`. Defaults to `false`.
  - `ignoreUndefined` <boolean> If `true`, specifies that the default writer will not output the return value of a command if it evaluates to `undefined`. Defaults to `false`.
  - `writer` <Function> The function to invoke to format the output of each command before writing to `output`. Defaults to util.inspect().
  - `completer` <Function> An optional function used for custom Tab auto completion. See readline.InterfaceCompleter for an example.
  - `replMode` <symbol> A flag that specifies whether the default evaluator executes all JavaScript commands in strict mode or default (sloppy) mode. Acceptable values are:
    - `repl.REPL_MODE_SLOPPY` - evaluates expressions in sloppy mode.
    - `repl.REPL_MODE_STRICT` - evaluates expressions in strict mode. This is equivalent to prefacing every repl statement with `'use strict'`.
    - `repl.REPL_MODE_MAGIC` - This value is **deprecated**, since enhanced spec compliance in V8 has rendered magic mode unnecessary. It is now equivalent to `repl.REPL_MODE_SLOPPY` (documented above).

- `breakEvalOnSigint` - Stop evaluating the current piece of code when `SIGINT` is received, i.e. `Ctrl+C` is pressed. This cannot be used together with a custom `eval` function. Defaults to `false`.

The `repl.start()` method creates and starts a `repl.REPLServer` instance.

If `options` is a string, then it specifies the input prompt:

```
const repl = require('repl');

// a Unix style prompt
repl.start('$ ');
```

The Node.js REPL

Node.js itself uses the `repl` module to provide its own interactive interface for executing JavaScript. This can be used by executing the Node.js binary without passing any arguments (or by passing the `-i` argument):

```
$ node
> const a = [1, 2, 3];
undefined
> a
[ 1, 2, 3 ]
> a.forEach((v) => {
...   console.log(v);
...   });
1
2
3
```

Environment Variable Options

Various behaviors of the Node.js REPL can be customized using the following environment variables:

- `NODE_REPL_HISTORY` - When a valid path is given, persistent REPL history will be saved to the specified file rather than `.node_repl_history` in the user's home directory. Setting this value to `''` will disable persistent REPL history. Whitespace will be trimmed from the value.
- `NODE_REPL_HISTORY_SIZE` - Defaults to `1000`. Controls how many lines of history will be persisted if history is available. Must be a positive number.
- `NODE_REPL_MODE` - May be any of `sloppy`, `strict`, or `magic`. Defaults to `sloppy`, which will allow non-strict mode code to be run. `magic` is **deprecated** and treated as an alias of `sloppy`.

Persistent History

By default, the Node.js REPL will persist history between `node` REPL sessions by saving inputs to a `.node_repl_history` file located in the user's home directory. This can be disabled by setting the environment variable `NODE_REPL_HISTORY=''`.

NODE_REPL_HISTORY_FILE

Added in: v2.0.0   Deprecated since: v3.0.0

Stability: 0 - Deprecated: Use `NODE_REPL_HISTORY` instead.

Previously in Node.js/io.js v2.x, REPL history was controlled by using a `NODE_REPL_HISTORY_FILE` environment variable, and the history was saved in JSON format. This variable has now been deprecated, and the old JSON REPL history file will be automatically converted to a simplified plain text format. This new file will be saved to either the user's home directory, or a directory defined by the `NODE_REPL_HISTORY` variable, as documented in the Environment Variable Options.

Using the Node.js REPL with advanced line-editors

For advanced line-editors, start Node.js with the environment variable `NODE_NO_READLINE=1`. This will start the main and debugger REPL in canonical terminal settings, which will allow use with `rlwrap`.

For example, the following can be added to a `.bashrc` file:

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

Starting multiple REPL instances against a single running instance

It is possible to create and run multiple REPL instances against a single running instance of Node.js that share a single `global` object but have separate I/O interfaces.

The following example, for instance, provides separate REPLs on `stdin`, a Unix socket, and a TCP socket:

```
const net = require('net');
const repl = require('repl');
let connections = 0;

repl.start({
  prompt: 'Node.js via stdin> ',
  input: process.stdin,
  output: process.stdout
});

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via Unix socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen('/tmp/node-repl-sock');

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via TCP socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen(5001);
```

Running this application from the command line will start a REPL on stdin. Other REPL clients may connect through the Unix socket or TCP socket. `telnet`, for instance, is useful for connecting to TCP sockets, while `socat` can be used to connect to both Unix and TCP sockets.

By starting a REPL from a Unix socket-based server instead of stdin, it is possible to connect to a long-running Node.js process without restarting it.

For an example of running a "full-featured" (`terminal`) REPL over a `net.Server` and `net.Socket` instance, see: https://gist.github.com/2209310

For an example of running a REPL instance over curl(1), see: https://gist.github.com/2053342

Stream

Stability: 2 - Stable

A stream is an abstract interface for working with streaming data in Node.js. The `stream` module provides a base API that makes it easy to build objects that implement the stream interface.

There are many stream objects provided by Node.js. For instance, a request to an HTTP server and `process.stdout` are both stream instances.

Streams can be readable, writable, or both. All streams are instances of `EventEmitter`.

The `stream` module can be accessed using:

```
const stream = require('stream');
```

While it is important to understand how streams work, the `stream` module itself is most useful for developers that are creating new types of stream instances. Developers who are primarily *consuming* stream objects will rarely need to use the `stream` module directly.

Organization of this Document

This document is divided into two primary sections with a third section for additional notes. The first section explains the elements of the stream API that are required to *use* streams within an application. The second section explains the elements of the API that are required to *implement* new types of streams.

Types of Streams

There are four fundamental stream types within Node.js:

- Readable - streams from which data can be read (for example `fs.createReadStream()`).
- Writable - streams to which data can be written (for example `fs.createWriteStream()`).
- Duplex - streams that are both Readable and Writable (for example `net.Socket`).
- Transform - Duplex streams that can modify or transform the data as it is written and read (for example `zlib.createDeflate()`).

Object Mode

All streams created by Node.js APIs operate exclusively on strings and `Buffer` (or `Uint8Array`) objects. It is possible, however, for stream implementations to work with other types of JavaScript values (with the exception of `null`, which serves a special purpose within streams). Such streams are considered to operate in "object mode".

Stream instances are switched into object mode using the `objectMode` option when the stream is created. Attempting to switch an existing stream into object mode is not safe.

Buffering

Both Writable and Readable streams will store data in an internal buffer that can be retrieved using `writable.writableBuffer` or `readable.readableBuffer`, respectively.

The amount of data potentially buffered depends on the `highWaterMark` option passed into the streams constructor. For normal streams, the `highWaterMark` option specifies a total number of bytes. For streams operating in object mode, the `highWaterMark` specifies a total number of objects.

Data is buffered in Readable streams when the implementation calls `stream.push(chunk)`. If the consumer of the Stream does not call `stream.read()`, the data will sit in the internal queue until it is consumed.

Once the total size of the internal read buffer reaches the threshold specified by `highWaterMark`, the stream will temporarily stop reading data from the underlying resource until the data currently buffered can be consumed (that is, the stream will stop calling the internal `readable._read()` method that is used to fill the read buffer).

Data is buffered in Writable streams when the writable.write(chunk) method is called repeatedly. While the total size of the internal write buffer is below the threshold set by `highWaterMark`, calls to `writable.write()` will return `true`. Once the size of the internal buffer reaches or exceeds the `highWaterMark`, `false` will be returned.

A key goal of the `stream` API, particularly the stream.pipe() method, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory.

Because Duplex and Transform streams are both Readable and Writable, each maintain *two* separate internal buffers used for reading and writing, allowing each side to operate independently of the other while maintaining an appropriate and efficient flow of data. For example, net.Socket instances are Duplex streams whose Readable side allows consumption of data received*from* the socket and whose Writable side allows writing data *to* the socket. Because data may be written to the socket at a faster or slower rate than data is received, it is important for each side to operate (and buffer) independently of the other.

API for Stream Consumers

Almost all Node.js applications, no matter how simple, use streams in some manner. The following is an example of using streams in a Node.js application that implements an HTTP server:

```
const http = require('http');

const server = http.createServer((req, res) => {
  // req is an http.IncomingMessage, which is a Readable Stream
  // res is an http.ServerResponse, which is a Writable Stream

  let body = '';
  // Get the data as utf8 strings.
  // If an encoding is not set, Buffer objects will be received.
  req.setEncoding('utf8');

  // Readable streams emit 'data' events once a listener is added
  req.on('data', (chunk) => {
    body += chunk;
  });

  // the end event indicates that the entire body has been received
  req.on('end', () => {
    try {
      const data = JSON.parse(body);
      // write back something interesting to the user:
      res.write(typeof data);
      res.end();
    } catch (er) {
      // uh oh! bad json!
      res.statusCode = 400;
      return res.end(`error: ${er.message}`);
    }
  });
});

server.listen(1337);

// $ curl localhost:1337 -d "{}"
// object
// $ curl localhost:1337 -d "\"foo\""
// string
// $ curl localhost:1337 -d "not json"
// error: Unexpected token o in JSON at position 1
```

Writable streams (such as res in the example) expose methods such as write() and end() that are used to write data onto the stream.

Readable streams use the EventEmitter API for notifying application code when data is available to be read off the stream. That available data can be read from the stream in multiple ways.

Both Writable and Readable streams use the EventEmitter API in various ways to communicate the current state of the stream.

Duplex and Transform streams are both Writable and Readable.

Applications that are either writing data to or consuming data from a stream are not required to implement the stream interfaces directly and will generally have no reason to call require('stream').

Developers wishing to implement new types of streams should refer to the section API for Stream Implementers.

Writable Streams

Writable streams are an abstraction for a *destination* to which data is written.

Examples of Writable streams include:

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr

Some of these examples are actually Duplex streams that implement the Writable interface.

All Writable streams implement the interface defined by the stream.Writable class.

While specific instances of Writable streams may differ in various ways, all Writable streams follow the same fundamental usage pattern as illustrated in the example below:

```
const myStream = getWritableStreamSomehow();
myStream.write('some data');
myStream.write('some more data');
myStream.end('done writing data');
```

Class: stream.Writable

Added in: v0.9.4

Event: 'close'

Added in: v0.9.4

The 'close' event is emitted when the stream and any of its underlying resources (a file descriptor, for example) have been closed. The event indicates that no more events will be emitted, and no further computation will occur.

Not all Writable streams will emit the 'close' event.

Event: 'drain'

Added in: v0.9.4

If a call to stream.write(chunk) returns false, the 'drain' event will be emitted when it is appropriate to resume writing data to the stream.

```
// Write the data to the supplied writable stream one million times.
// Be attentive to back-pressure.
function writeOneMillionTimes(writer, data, encoding, callback) {
  let i = 1000000;
  write();
  function write() {
    let ok = true;
    do {
      i--;
      if (i === 0) {
        // last time!
        writer.write(data, encoding, callback);
      } else {
        // see if we should continue, or wait
        // don't pass the callback, because we're not done yet.
        ok = writer.write(data, encoding);
      }
    } while (i > 0 && ok);
    if (i > 0) {
      // had to stop early!
      // write some more once it drains
      writer.once('drain', write);
    }
  }
}
```

Event: 'error'

Added in: v0.9.4

- <Error>

The `'error'` event is emitted if an error occurred while writing or piping data. The listener callback is passed a single `Error` argument when called.

The stream is not closed when the `'error'` event is emitted.

Event: 'finish'

Added in: v0.9.4

The `'finish'` event is emitted after the stream.end() method has been called, and all data has been flushed to the underlying system.

```
const writer = getWritableStreamSomehow();
for (let i = 0; i < 100; i++) {
  writer.write(`hello, #${i}!\n`);
}
writer.end('This is the end\n');
writer.on('finish', () => {
  console.error('All writes are now complete.');
});
```

Event: 'pipe'

Added in: v0.9.4

- `src` <stream.Readable> source stream that is piping to this writable

The `'pipe'` event is emitted when the stream.pipe() method is called on a readable stream, adding this writable to its set of destinations.

```
const writer = getWritableStreamSomehow();
const reader = getReadableStreamSomehow();
writer.on('pipe', (src) => {
  console.error('something is piping into the writer');
  assert.equal(src, reader);
});
reader.pipe(writer);
```

Event: 'unpipe'

Added in: v0.9.4

- `src` <stream.Readable> The source stream that unpiped this writable

The `'unpipe'` event is emitted when the stream.unpipe() method is called on a Readable stream, removing this Writable from its set of destinations.

This is also emitted in case this Writable stream emits an error when a Readable stream pipes into it.

```
const writer = getWritableStreamSomehow();
const reader = getReadableStreamSomehow();
writer.on('unpipe', (src) => {
  console.error('Something has stopped piping into the writer.');
  assert.equal(src, reader);
});
reader.pipe(writer);
reader.unpipe(writer);
```

writable.cork()

Added in: v0.11.2

The `writable.cork()` method forces all written data to be buffered in memory. The buffered data will be flushed when either the stream.uncork() or stream.end() methods are called.

The primary intent of `writable.cork()` is to avoid a situation where writing many small chunks of data to a stream do not cause a backup in the internal buffer that would have an adverse impact on performance. In such situations, implementations that implement the `writable._writev()` method can perform buffered writes in a more optimized manner.

See also: writable.uncork().

writable.end([chunk][, encoding][, callback])

▸ History

- `chunk` <string> | <Buffer> | <Uint8Array> | <any> Optional data to write. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any JavaScript value other than `null`.
- `encoding` <string> The encoding, if `chunk` is a string
- `callback` <Function> Optional callback for when the stream is finished

Calling the `writable.end()` method signals that no more data will be written to the Writable. The optional `chunk` and `encoding` arguments allow one final additional chunk of data to be written immediately before closing the stream. If provided, the optional `callback` function is attached as a listener for the 'finish' event.

Calling the stream.write() method after calling stream.end() will raise an error.

```
// write 'hello, ' and then end with 'world!'
const fs = require('fs');
const file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// writing more now is not allowed!
```

writable.setDefaultEncoding(encoding)

▸ History

- `encoding` <string> The new default encoding
- Returns: <this>

The `writable.setDefaultEncoding()` method sets the default `encoding` for a Writable stream.

writable.uncork()

Added in: v0.11.2

The `writable.uncork()` method flushes all data buffered since stream.cork() was called.

When using writable.cork() and `writable.uncork()` to manage the buffering of writes to a stream, it is recommended that calls to `writable.uncork()` be deferred using `process.nextTick()`. Doing so allows batching of all `writable.write()` calls that occur within a given Node.js event loop phase.

```
stream.cork();
stream.write('some ');
stream.write('data ');
process.nextTick(() => stream.uncork());
```

If the writable.cork() method is called multiple times on a stream, the same number of calls to `writable.uncork()` must be called to flush the buffered data.

```
stream.cork();
stream.write('some ');
stream.cork();
stream.write('data ');
process.nextTick(() => {
  stream.uncork();
  // The data will not be flushed until uncork() is called a second time.
  stream.uncork();
});
```

See also: writable.cork().

writable.writableHighWaterMark

Added in: v9.3.0

Return the value of `highWaterMark` passed when constructing this `Writable`.

writable.writableLength

Added in: v9.4.0

This property contains the number of bytes (or objects) in the queue ready to be written. The value provides introspection data regarding the status of the `highWaterMark`.

writable.write(chunk[, encoding][, callback])

▸ History

- chunk <string> | <Buffer> | <Uint8Array> | <any> Optional data to write. For streams not operating in object mode, chunk must be a string, Buffer or Uint8Array. For object mode streams, chunk may be any JavaScript value other than null.
- encoding <string> The encoding, if chunk is a string
- callback <Function> Callback for when this chunk of data is flushed
- Returns: <boolean> false if the stream wishes for the calling code to wait for the 'drain' event to be emitted before continuing to write additional data; otherwise true.

The writable.write() method writes some data to the stream, and calls the supplied callback once the data has been fully handled. If an error occurs, the callback *may or may not* be called with the error as its first argument. To reliably detect write errors, add a listener for the 'error' event.

The return value is true if the internal buffer is less than the highWaterMark configured when the stream was created after admitting chunk. If false is returned, further attempts to write data to the stream should stop until the 'drain' event is emitted.

While a stream is not draining, calls to write() will buffer chunk, and return false. Once all currently buffered chunks are drained (accepted for delivery by the operating system), the 'drain' event will be emitted. It is recommended that once write() returns false, no more chunks be written until the 'drain' event is emitted. While calling write() on a stream that is not draining is allowed, Node.js will buffer all written chunks until maximum memory usage occurs, at which point it will abort unconditionally. Even before it aborts, high memory usage will cause poor garbage collector performance and high RSS (which is not typically released back to the system, even after the memory is no longer required). Since TCP sockets may never drain if the remote peer does not read the data, writing a socket that is not draining may lead to a remotely exploitable vulnerability.

Writing data while the stream is not draining is particularly problematic for a Transform, because the Transform streams are paused by default until they are piped or an 'data' or 'readable' event handler is added.

If the data to be written can be generated or fetched on demand, it is recommended to encapsulate the logic into a Readable and use stream.pipe(). However, if calling write() is preferred, it is possible to respect backpressure and avoid memory issues using the 'drain' event:

```
function write(data, cb) {
  if (!stream.write(data)) {
    stream.once('drain', cb);
  } else {
    process.nextTick(cb);
  }
}

// Wait for cb to be called before doing any other write.
write('hello', () => {
  console.log('write completed, do more writes now');
});
```

A Writable stream in object mode will always ignore the encoding argument.

writable.destroy([error])

Added in: v8.0.0

- Returns: <this>

Destroy the stream, and emit the passed error. After this call, the writable stream has ended. Implementors should not override this method, but instead implement writable._destroy.

Readable Streams

Readable streams are an abstraction for a *source* from which data is consumed.

Examples of Readable streams include:

- HTTP responses, on the client

- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

All Readable streams implement the interface defined by the `stream.Readable` class.

Two Modes

Readable streams effectively operate in one of two modes: flowing and paused.

When in flowing mode, data is read from the underlying system automatically and provided to an application as quickly as possible using events via the EventEmitter interface.

In paused mode, the stream.read() method must be called explicitly to read chunks of data from the stream.

All Readable streams begin in paused mode but can be switched to flowing mode in one of the following ways:

- Adding a 'data' event handler.
- Calling the stream.resume() method.
- Calling the stream.pipe() method to send the data to a Writable.

The Readable can switch back to paused mode using one of the following:

- If there are no pipe destinations, by calling the stream.pause() method.
- If there are pipe destinations, by removing all pipe destinations. Multiple pipe destinations may be removed by calling the stream.unpipe() method.

The important concept to remember is that a Readable will not generate data until a mechanism for either consuming or ignoring that data is provided. If the consuming mechanism is disabled or taken away, the Readable will *attempt* to stop generating the data.

For backwards compatibility reasons, removing 'data' event handlers will **not** automatically pause the stream. Also, if there are piped destinations, then calling stream.pause() will not guarantee that the stream will *remain* paused once those destinations drain and ask for more data.

If a Readable is switched into flowing mode and there are no consumers available to handle the data, that data will be lost. This can occur, for instance, when the `readable.resume()` method is called without a listener attached to the `'data'` event, or when a `'data'` event handler is removed from the stream.

Three States

The "two modes" of operation for a Readable stream are a simplified abstraction for the more complicated internal state management that is happening within the Readable stream implementation.

Specifically, at any given point in time, every Readable is in one of three possible states:

- `readable.readableFlowing = null`
- `readable.readableFlowing = false`
- `readable.readableFlowing = true`

When `readable.readableFlowing` is `null`, no mechanism for consuming the streams data is provided so the stream will not generate its data. While in this state, attaching a listener for the `'data'` event, calling the `readable.pipe()` method, or calling the `readable.resume()` method will switch `readable.readableFlowing` to `true`, causing the Readable to begin actively emitting events as data is generated.

Calling `readable.pause()`, `readable.unpipe()`, or receiving "back pressure" will cause the `readable.readableFlowing` to be set as `false`, temporarily halting the flowing of events but *not* halting the generation of data. While in this state, attaching a listener for the `'data'` event would not cause `readable.readableFlowing` to switch to `true`.

```
const { PassThrough, Writable } = require('stream');
const pass = new PassThrough();
const writable = new Writable();

pass.pipe(writable);
pass.unpipe(writable);
// readableFlowing is now false

pass.on('data', (chunk) => { console.log(chunk.toString()); });
pass.write('ok'); // will not emit 'data'
pass.resume(); // must be called to make 'data' being emitted
```

While `readable.readableFlowing` is `false`, data may be accumulating within the streams internal buffer.

Choose One

The Readable stream API evolved across multiple Node.js versions and provides multiple methods of consuming stream data. In general, developers should choose *one* of the methods of consuming data and *should never* use multiple methods to consume data from a single stream.

Use of the `readable.pipe()` method is recommended for most users as it has been implemented to provide the easiest way of consuming stream data. Developers that require more fine-grained control over the transfer and generation of data can use the EventEmitter and `readable.pause()` / `readable.resume()` APIs.

Class: stream.Readable

Added in: v0.9.4

Event: 'close'

Added in: v0.9.4

The `'close'` event is emitted when the stream and any of its underlying resources (a file descriptor, for example) have been closed. The event indicates that no more events will be emitted, and no further computation will occur.

Not all Readable streams will emit the `'close'` event.

Event: 'data'

Added in: v0.9.4

- `chunk` <Buffer> | <string> | <any> The chunk of data. For streams that are not operating in object mode, the chunk will be either a string or `Buffer`. For streams that are in object mode, the chunk can be any JavaScript value other than `null`.

The `'data'` event is emitted whenever the stream is relinquishing ownership of a chunk of data to a consumer. This may occur whenever the stream is switched in flowing mode by calling `readable.pipe()`, `readable.resume()`, or by attaching a listener callback to the `'data'` event. The `'data'` event will also be emitted whenever the `readable.read()` method is called and a chunk of data is available to be returned.

Attaching a `'data'` event listener to a stream that has not been explicitly paused will switch the stream into flowing mode. Data will then be passed as soon as it is available.

The listener callback will be passed the chunk of data as a string if a default encoding has been specified for the stream using the `readable.setEncoding()` method; otherwise the data will be passed as a `Buffer`.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
```

Event: 'end'

Added in: v0.9.4

The 'end' event is emitted when there is no more data to be consumed from the stream.

The 'end' event **will not be emitted** unless the data is completely consumed. This can be accomplished by switching the stream into flowing mode, or by calling stream.read() repeatedly until all data has been consumed.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
readable.on('end', () => {
  console.log('There will be no more data.');
});
```

Event: 'error'

Added in: v0.9.4

- <Error>

The 'error' event may be emitted by a Readable implementation at any time. Typically, this may occur if the underlying stream is unable to generate data due to an underlying internal failure, or when a stream implementation attempts to push an invalid chunk of data.

The listener callback will be passed a single Error object.

Event: 'readable'

Added in: v0.9.4

The 'readable' event is emitted when there is data available to be read from the stream. In some cases, attaching a listener for the 'readable' event will cause some amount of data to be read into an internal buffer.

```
const readable = getReadableStreamSomehow();
readable.on('readable', () => {
  // there is some data to read now
});
```

The 'readable' event will also be emitted once the end of the stream data has been reached but before the 'end' event is emitted.

Effectively, the 'readable' event indicates that the stream has new information: either new data is available or the end of the stream has been reached. In the former case, stream.read() will return the available data. In the latter case, stream.read() will return null. For instance, in the following example, foo.txt is an empty file:

```
const fs = require('fs');
const rr = fs.createReadStream('foo.txt');
rr.on('readable', () => {
  console.log(`readable: ${rr.read()}`);
});
rr.on('end', () => {
  console.log('end');
});
```

The output of running this script is:

```
$ node test.js
readable: null
end
```

In general, the readable.pipe() and 'data' event mechanisms are easier to understand than the 'readable' event. However, handling 'readable' might result in increased throughput.

readable.isPaused()

Added in: v0.11.14

- Returns: <boolean>

The `readable.isPaused()` method returns the current operating state of the Readable. This is used primarily by the mechanism that underlies the `readable.pipe()` method. In most typical cases, there will be no reason to use this method directly.

```
const readable = new stream.Readable();

readable.isPaused(); // === false
readable.pause();
readable.isPaused(); // === true
readable.resume();
readable.isPaused(); // === false
```

readable.pause()

Added in: v0.9.4

- Returns: <this>

The `readable.pause()` method will cause a stream in flowing mode to stop emitting `'data'` events, switching out of flowing mode. Any data that becomes available will remain in the internal buffer.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
  readable.pause();
  console.log('There will be no additional data for 1 second.');
  setTimeout(() => {
    console.log('Now data will start flowing again.');
    readable.resume();
  }, 1000);
});
```

readable.pipe(destination[, options])

Added in: v0.9.4

- `destination` <stream.Writable> The destination for writing data
- `options` <Object> Pipe options
  - `end` <boolean> End the writer when the reader ends. Defaults to `true`.

The `readable.pipe()` method attaches a Writable stream to the `readable`, causing it to switch automatically into flowing mode and push all of its data to the attached Writable. The flow of data will be automatically managed so that the destination Writable stream is not overwhelmed by a faster Readable stream.

The following example pipes all of the data from the `readable` into a file named `file.txt`:

```
const fs = require('fs');
const readable = getReadableStreamSomehow();
const writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt'
readable.pipe(writable);
```

It is possible to attach multiple Writable streams to a single Readable stream.

The `readable.pipe()` method returns a reference to the *destination* stream making it possible to set up chains of piped streams:

```
const fs = require('fs');
const r = fs.createReadStream('file.txt');
const z = zlib.createGzip();
const w = fs.createWriteStream('file.txt.gz');
r.pipe(z).pipe(w);
```

By default, `stream.end()` is called on the destination Writable stream when the source Readable stream emits `'end'`, so that the destination is no longer writable. To disable this default behavior, the `end` option can be passed as `false`, causing the destination stream to remain open, as illustrated in the following example:

```
reader.pipe(writer, { end: false });
reader.on('end', () => {
  writer.end('Goodbye\n');
});
```

One important caveat is that if the Readable stream emits an error during processing, the Writable destination*is not closed* automatically. If an error occurs, it will be necessary to *manually* close each stream in order to prevent memory leaks.

The `process.stderr` and `process.stdout` Writable streams are never closed until the Node.js process exits, regardless of the specified options.

readable.readableHighWaterMark

Added in: v9.3.0

Return the value of `highWaterMark` passed when constructing this `Readable`.

readable.read([size])

Added in: v0.9.4

- `size` <number> Optional argument to specify how much data to read.
- Return <string> | <Buffer> | <null>

The `readable.read()` method pulls some data out of the internal buffer and returns it. If no data available to be read, `null` is returned. By default, the data will be returned as a `Buffer` object unless an encoding has been specified using the `readable.setEncoding()` method or the stream is operating in object mode.

The optional `size` argument specifies a specific number of bytes to read. If `size` bytes are not available to be read, `null` will be returned *unless* the stream has ended, in which case all of the data remaining in the internal buffer will be returned.

If the `size` argument is not specified, all of the data contained in the internal buffer will be returned.

The `readable.read()` method should only be called on Readable streams operating in paused mode. In flowing mode, `readable.read()` is called automatically until the internal buffer is fully drained.

```
const readable = getReadableStreamSomehow();
readable.on('readable', () => {
  let chunk;
  while (null !== (chunk = readable.read())) {
    console.log(`Received ${chunk.length} bytes of data.`);
  }
});
```

A Readable stream in object mode will always return a single item from a call to `readable.read(size)`, regardless of the value of the `size` argument.

If the `readable.read()` method returns a chunk of data, a `'data'` event will also be emitted.

Calling `stream.read([size])` after the `'end'` event has been emitted will return `null`. No runtime error will be raised.

readable.readableLength

Added in: v9.4.0

This property contains the number of bytes (or objects) in the queue ready to be read. The value provides introspection data regarding the status of the `highWaterMark`.

readable.resume()

Added in: v0.9.4

- Returns: <this>

The `readable.resume()` method causes an explicitly paused Readable stream to resume emitting `'data'` events, switching the stream into flowing mode.

The `readable.resume()` method can be used to fully consume the data from a stream without actually processing any of that data as illustrated in the following example:

```
getReadableStreamSomehow()
 .resume()
 .on('end', () => {
   console.log('Reached the end, but did not read anything.');
 });
```

readable.setEncoding(encoding)

Added in: v0.9.4

- `encoding` <string> The encoding to use.
- Returns: <this>

The `readable.setEncoding()` method sets the character encoding for data read from the Readable stream.

By default, no encoding is assigned and stream data will be returned as `Buffer` objects. Setting an encoding causes the stream data to be returned as strings of the specified encoding rather than as `Buffer` objects. For instance, calling `readable.setEncoding('utf8')` will cause the output data to be interpreted as UTF-8 data, and passed as strings. Calling `readable.setEncoding('hex')` will cause the data to be encoded in hexadecimal string format.

The Readable stream will properly handle multi-byte characters delivered through the stream that would otherwise become improperly decoded if simply pulled from the stream as `Buffer` objects.

```
const readable = getReadableStreamSomehow();
readable.setEncoding('utf8');
readable.on('data', (chunk) => {
  assert.equal(typeof chunk, 'string');
  console.log('got %d characters of string data', chunk.length);
});
```

readable.unpipe([destination])

Added in: v0.9.4

- `destination` <stream.Writable> Optional specific stream to unpipe

The `readable.unpipe()` method detaches a Writable stream previously attached using the `stream.pipe()` method.

If the `destination` is not specified, then *all* pipes are detached.

If the `destination` is specified, but no pipe is set up for it, then the method does nothing.

```
const fs = require('fs');
const readable = getReadableStreamSomehow();
const writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt',
// but only for the first second
readable.pipe(writable);
setTimeout(() => {
  console.log('Stop writing to file.txt');
  readable.unpipe(writable);
  console.log('Manually close the file stream');
  writable.end();
}, 1000);
```

readable.unshift(chunk)

▸ History

- `chunk` <Buffer> | <Uint8Array> | <string> | <any> Chunk of data to unshift onto the read queue. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any JavaScript value other than `null`.

The `readable.unshift()` method pushes a chunk of data back into the internal buffer. This is useful in certain situations where a stream is being consumed by code that needs to "un-consume" some amount of data that it has optimistically pulled out of the source, so that the data can be passed on to some other party.

The `stream.unshift(chunk)` method cannot be called after the `'end'` event has been emitted or a runtime error will be thrown.

Developers using `stream.unshift()` often should consider switching to use of a Transform stream instead. See the API for Stream Implementers section for more information.

```
// Pull off a header delimited by \n\n
// use unshift() if we get too much
// Call the callback with (error, header, stream)
const { StringDecoder } = require('string_decoder');
function parseHeader(stream, callback) {
  stream.on('error', callback);
  stream.on('readable', onReadable);
  const decoder = new StringDecoder('utf8');
  let header = '';
  function onReadable() {
    let chunk;
    while (null !== (chunk = stream.read())) {
      const str = decoder.write(chunk);
      if (str.match(/\n\n/)) {
        // found the header boundary
        const split = str.split(/\n\n/);
        header += split.shift();
        const remaining = split.join('\n\n');
        const buf = Buffer.from(remaining, 'utf8');
        stream.removeListener('error', callback);
        // remove the readable listener before unshifting
        stream.removeListener('readable', onReadable);
        if (buf.length)
          stream.unshift(buf);
        // now the body of the message can be read from the stream.
        callback(null, header, stream);
      } else {
        // still reading the header.
        header += str;
      }
    }
  }
}
```

Unlike `stream.push(chunk)`, `stream.unshift(chunk)` will not end the reading process by resetting the internal reading state of the stream. This can cause unexpected results if `readable.unshift()` is called during a read (i.e. from within a `stream._read()` implementation on a custom stream). Following the call to `readable.unshift()` with an immediate `stream.push('')` will reset the reading state appropriately, however it is best to simply avoid calling `readable.unshift()` while in the process of performing a read.

readable.wrap(stream)

Added in: v0.9.4

- `stream` <Stream> An "old style" readable stream

Versions of Node.js prior to v0.10 had streams that did not implement the entire `stream` module API as it is currently defined. (See Compatibility for more information.)

When using an older Node.js library that emits `'data'` events and has a `stream.pause()` method that is advisory only, the `readable.wrap()` method can be used to create a Readable stream that uses the old stream as its data source.

It will rarely be necessary to use `readable.wrap()` but the method has been provided as a convenience for interacting with older Node.js applications and libraries.

```js
const { OldReader } = require('./old-api-module.js');
const { Readable } = require('stream');
const oreader = new OldReader();
const myReader = new Readable().wrap(oreader);

myReader.on('readable', () => {
  myReader.read(); // etc.
});
```

readable.destroy([error])

Added in: v8.0.0

Destroy the stream, and emit `'error'`. After this call, the readable stream will release any internal resources. Implementors should not override this method, but instead implement `readable._destroy`.

Duplex and Transform Streams

Class: stream.Duplex

▸ History

Duplex streams are streams that implement both the Readable and Writable interfaces.

Examples of Duplex streams include:

- TCP sockets
- zlib streams
- crypto streams

Class: stream.Transform

Added in: v0.9.4

Transform streams are Duplex streams where the output is in some way related to the input. Like all Duplex streams, Transform streams implement both the Readable and Writable interfaces.

Examples of Transform streams include:

- zlib streams
- crypto streams

transform.destroy([error])

Added in: v8.0.0

Destroy the stream, and emit `'error'`. After this call, the transform stream would release any internal resources. implementors should not override this method, but instead implement `readable._destroy`. The default implementation of `_destroy` for `Transform` also emit `'close'`.

API for Stream Implementers

The `stream` module API has been designed to make it possible to easily implement streams using JavaScript's prototypal inheritance model.

First, a stream developer would declare a new JavaScript class that extends one of the four basic stream classes (`stream.Writable`, `stream.Readable`, `stream.Duplex`, or `stream.Transform`), making sure they call the appropriate parent class constructor:

```
const { Writable } = require('stream');

class MyWritable extends Writable {
  constructor(options) {
    super(options);
    // ...
  }
}
```

The new stream class must then implement one or more specific methods, depending on the type of stream being created, as detailed in the chart below:

Use-case	Class	Method(s) to implement
Reading only	Readable	`_read`
Writing only	Writable	`_write` , `_writev` , `_final`
Reading and writing	Duplex	`_read` , `_write` , `_writev` , `_final`
Operate on written data, then read the result	Transform	`_transform` , `_flush` , `_final`

The implementation code for a stream should *never* call the "public" methods of a stream that are intended for use by consumers (as described in the API for Stream Consumers section). Doing so may lead to adverse side effects in application code consuming the stream.

Simplified Construction

Added in: v1.2.0

For many simple cases, it is possible to construct a stream without relying on inheritance. This can be accomplished by directly creating instances of the `stream.Writable` , `stream.Readable` , `stream.Duplex` or `stream.Transform` objects and passing appropriate methods as constructor options.

```
const { Writable } = require('stream');

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    // ...
  }
});
```

Implementing a Writable Stream

The `stream.Writable` class is extended to implement a Writable stream.

Custom Writable streams *must* call the `new stream.Writable([options])` constructor and implement the `writable._write()` method. The `writable._writev()` method *may* also be implemented.

Constructor: new stream.Writable(options)

- `options` <Object>
  - `highWaterMark` <number> Buffer level when `stream.write()` starts returning `false` . Defaults to `16384` (16kb), or `16` for `objectMode` streams.
  - `decodeStrings` <boolean> Whether or not to decode strings into Buffers before passing them to `stream._write()` . Defaults to `true`
  - `objectMode` <boolean> Whether or not the `stream.write(anyObj)` is a valid operation. When set, it becomes possible to write JavaScript values other than string, `Buffer` or `Uint8Array` if supported by the stream implementation. Defaults to `false`

- `write` <Function> Implementation for the stream._write() method.
- `writev` <Function> Implementation for the stream._writev() method.
- `destroy` <Function> Implementation for the stream._destroy() method.
- `final` <Function> Implementation for the stream._final() method.

```
const { Writable } = require('stream');

class MyWritable extends Writable {
  constructor(options) {
    // Calls the stream.Writable() constructor
    super(options);
    // ...
  }
}
```

Or, when using pre-ES6 style constructors:

```
const { Writable } = require('stream');
const util = require('util');

function MyWritable(options) {
  if (!(this instanceof MyWritable))
    return new MyWritable(options);
  Writable.call(this, options);
}
util.inherits(MyWritable, Writable);
```

Or, using the Simplified Constructor approach:

```
const { Writable } = require('stream');

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    // ...
  },
  writev(chunks, callback) {
    // ...
  }
});
```

writable._write(chunk, encoding, callback)

- `chunk` <Buffer> | <string> | <any> The chunk to be written. Will **always** be a buffer unless the `decodeStrings` option was set to `false` or the stream is operating in object mode.
- `encoding` <string> If the chunk is a string, then `encoding` is the character encoding of that string. If chunk is a `Buffer`, or if the stream is operating in object mode, `encoding` may be ignored.
- `callback` <Function> Call this function (optionally with an error argument) when processing is complete for the supplied chunk.

All Writable stream implementations must provide a writable._write() method to send data to the underlying resource.

Transform streams provide their own implementation of the writable._write().

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal Writable class methods only.

The `callback` method must be called to signal either that the write completed successfully or failed with an error. The first argument passed to

the `callback` must be the `Error` object if the call failed or `null` if the write succeeded.

All calls to `writable.write()` that occur between the time `writable._write()` is called and the `callback` is called will cause the written data to be buffered. When the `callback` is invoked, the stream might emit a `'drain'` event. If a stream implementation is capable of processing multiple chunks of data at once, the `writable._writev()` method should be implemented.

If the `decodeStrings` property is set in the constructor options, then `chunk` may be a string rather than a Buffer, and `encoding` will indicate the character encoding of the string. This is to support implementations that have an optimized handling for certain string data encodings. If the `decodeStrings` property is explicitly set to `false`, the `encoding` argument can be safely ignored, and `chunk` will remain the same object that is passed to `.write()`.

The `writable._write()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

writable._writev(chunks, callback)

- `chunks` <Array> The chunks to be written. Each chunk has following format: `{ chunk: ..., encoding: ... }`.
- `callback` <Function> A callback function (optionally with an error argument) to be invoked when processing is complete for the supplied chunks.

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal Writable class methods only.

The `writable._writev()` method may be implemented in addition to `writable._write()` in stream implementations that are capable of processing multiple chunks of data at once. If implemented, the method will be called with all chunks of data currently buffered in the write queue.

The `writable._writev()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

writable._destroy(err, callback)

Added in: v8.0.0

- `err` <Error> A possible error.
- `callback` <Function> A callback function that takes an optional error argument.

The `_destroy()` method is called by `writable.destroy()`. It can be overridden by child classes but it **must not** be called directly.

writable._final(callback)

Added in: v8.0.0

- `callback` <Function> Call this function (optionally with an error argument) when finished writing any remaining data.

The `_final()` method **must not** be called directly. It may be implemented by child classes, and if so, will be called by the internal Writable class methods only.

This optional function will be called before the stream closes, delaying the `finish` event until `callback` is called. This is useful to close resources or write buffered data before a stream ends.

Errors While Writing

It is recommended that errors occurring during the processing of the `writable._write()` and `writable._writev()` methods are reported by invoking the callback and passing the error as the first argument. This will cause an `'error'` event to be emitted by the Writable. Throwing an Error from within `writable._write()` can result in unexpected and inconsistent behavior depending on how the stream is being used. Using the callback ensures consistent and predictable handling of errors.

If a Readable stream pipes into a Writable stream when Writable emits an error, the Readable stream will be unpiped.

```
const { Writable } = require('stream');

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
      callback(new Error('chunk is invalid'));
    } else {
      callback();
    }
  }
});
```

An Example Writable Stream

The following illustrates a rather simplistic (and somewhat pointless) custom Writable stream implementation. While this specific Writable stream instance is not of any real particular usefulness, the example illustrates each of the required elements of a custom Writable stream instance:

```
const { Writable } = require('stream');

class MyWritable extends Writable {
  constructor(options) {
    super(options);
    // ...
  }

  _write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
      callback(new Error('chunk is invalid'));
    } else {
      callback();
    }
  }
}
```

Decoding buffers in a Writable Stream

Decoding buffers is a common task, for instance, when using transformers whose input is a string. This is not a trivial process when using multi-byte characters encoding, such as UTF-8. The following example shows how to decode multi-byte strings using `StringDecoder` and Writable.

```
const { Writable } = require('stream');
const { StringDecoder } = require('string_decoder');

class StringWritable extends Writable {
  constructor(options) {
    super(options);
    const state = this._writableState;
    this._decoder = new StringDecoder(state.defaultEncoding);
    this.data = '';
  }
  _write(chunk, encoding, callback) {
    if (encoding === 'buffer') {
      chunk = this._decoder.write(chunk);
    }
    this.data += chunk;
    callback();
  }
  _final(callback) {
    this.data += this._decoder.end();
    callback();
  }
}

const euro = [[0xE2, 0x82], [0xAC]].map(Buffer.from);
const w = new StringWritable();

w.write('currency: ');
w.write(euro[0]);
w.end(euro[1]);

console.log(w.data); // currency: €
```

Implementing a Readable Stream

The `stream.Readable` class is extended to implement a Readable stream.

Custom Readable streams *must* call the `new stream.Readable([options])` constructor and implement the `readable._read()` method.

new stream.Readable(options)

- `options` <Object>
  - `highWaterMark` <number> The maximum number of bytes to store in the internal buffer before ceasing to read from the underlying resource. Defaults to `16384` (16kb), or `16` for `objectMode` streams
  - `encoding` <string> If specified, then buffers will be decoded to strings using the specified encoding. Defaults to `null`
  - `objectMode` <boolean> Whether this stream should behave as a stream of objects. Meaning that stream.read(n) returns a single value instead of a Buffer of size n. Defaults to `false`
  - `read` <Function> Implementation for the stream._read() method.
  - `destroy` <Function> Implementation for the stream._destroy() method.

```
const { Readable } = require('stream');

class MyReadable extends Readable {
  constructor(options) {
    // Calls the stream.Readable(options) constructor
    super(options);
    // ...
  }
}
```

Or, when using pre-ES6 style constructors:

```
const { Readable } = require('stream');
const util = require('util');

function MyReadable(options) {
  if (!(this instanceof MyReadable))
    return new MyReadable(options);
  Readable.call(this, options);
}
util.inherits(MyReadable, Readable);
```

Or, using the Simplified Constructor approach:

```
const { Readable } = require('stream');

const myReadable = new Readable({
  read(size) {
    // ...
  }
});
```

readable._read(size)

- `size` <number> Number of bytes to read asynchronously

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal Readable class methods only.

All Readable stream implementations must provide an implementation of the `readable._read()` method to fetch data from the underlying resource.

When `readable._read()` is called, if data is available from the resource, the implementation should begin pushing that data into the read queue using the this.push(dataChunk) method. `_read()` should continue reading from the resource and pushing data until `readable.push()` returns `false`. Only when `_read()` is called again after it has stopped should it resume pushing additional data onto the queue.

Once the `readable._read()` method has been called, it will not be called again until the readable.push() method is called.

The `size` argument is advisory. For implementations where a "read" is a single operation that returns data can use the `size` argument to determine how much data to fetch. Other implementations may ignore this argument and simply provide data whenever it becomes available. There is no need to "wait" until `size` bytes are available before calling stream.push(chunk).

The `readable._read()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

readable._destroy(err, callback)

Added in: v8.0.0

- `err` <Error> A possible error.
- `callback` <Function> A callback function that takes an optional error argument.

The `_destroy()` method is called by `readable.destroy()`. It can be overridden by child classes but it **must not** be called directly.

readable.push(chunk[, encoding])

▸ History

- `chunk` <Buffer> | <Uint8Array> | <string> | <null> | <any> Chunk of data to push into the read queue. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any JavaScript value.
- `encoding` <string> Encoding of string chunks. Must be a valid Buffer encoding, such as `'utf8'` or `'ascii'`
- Returns: <boolean> `true` if additional chunks of data may continued to be pushed; `false` otherwise.

When `chunk` is a `Buffer`, `Uint8Array` or `string`, the `chunk` of data will be added to the internal queue for users of the stream to consume. Passing `chunk` as `null` signals the end of the stream (EOF), after which no more data can be written.

When the Readable is operating in paused mode, the data added with `readable.push()` can be read out by calling the `readable.read()` method when the `'readable'` event is emitted.

When the Readable is operating in flowing mode, the data added with `readable.push()` will be delivered by emitting a `'data'` event.

The `readable.push()` method is designed to be as flexible as possible. For example, when wrapping a lower-level source that provides some form of pause/resume mechanism, and a data callback, the low-level source can be wrapped by the custom Readable instance as illustrated in the following example:

```
// source is an object with readStop() and readStart() methods,
// and an `ondata` member that gets called when it has data, and
// an `onend` member that gets called when the data is over.

class SourceWrapper extends Readable {
  constructor(options) {
    super(options);

    this._source = getLowlevelSourceObject();

    // Every time there's data, push it into the internal buffer.
    this._source.ondata = (chunk) => {
      // if push() returns false, then stop reading from source
      if (!this.push(chunk))
        this._source.readStop();
    };

    // When the source ends, push the EOF-signaling `null` chunk
    this._source.onend = () => {
      this.push(null);
    };
  }
  // _read will be called when the stream wants to pull more data in
  // the advisory size argument is ignored in this case.
  _read(size) {
    this._source.readStart();
  }
}
```

The `readable.push()` method is intended be called only by Readable Implementers, and only from within the `readable._read()` method.

For streams not operating in object mode, if the `chunk` parameter of `readable.push()` is `undefined`, it will be treated as empty string or buffer. See `readable.push('')` for more information.

Errors While Reading

It is recommended that errors occurring during the processing of the `readable._read()` method are emitted using the `'error'` event rather than being thrown. Throwing an Error from within `readable._read()` can result in unexpected and inconsistent behavior depending on whether the stream is operating in flowing or paused mode. Using the `'error'` event ensures consistent and predictable handling of errors.

```
const { Readable } = require('stream');

const myReadable = new Readable({
  read(size) {
    if (checkSomeErrorCondition()) {
      process.nextTick(() => this.emit('error', err));
      return;
    }
    // do some work
  }
});
```

An Example Counting Stream

The following is a basic example of a Readable stream that emits the numerals from 1 to 1,000,000 in ascending order, and then ends.

```
const { Readable } = require('stream');

class Counter extends Readable {
  constructor(opt) {
    super(opt);
    this._max = 1000000;
    this._index = 1;
  }

  _read() {
    const i = this._index++;
    if (i > this._max)
      this.push(null);
    else {
      const str = String(i);
      const buf = Buffer.from(str, 'ascii');
      this.push(buf);
    }
  }
}
```

Implementing a Duplex Stream

A Duplex stream is one that implements both Readable and Writable, such as a TCP socket connection.

Because JavaScript does not have support for multiple inheritance, the `stream.Duplex` class is extended to implement a Duplex stream (as opposed to extending the `stream.Readable` *and* `stream.Writable` classes).

The `stream.Duplex` class prototypically inherits from `stream.Readable` and parasitically from `stream.Writable`, but `instanceof` will work properly for both base classes due to overriding `Symbol.hasInstance` on `stream.Writable`.

Custom Duplex streams *must* call the `new stream.Duplex([options])` constructor and implement *both* the `readable._read()` and `writable._write()` methods.

new stream.Duplex(options)

▶ History

- `options` <Object> Passed to both Writable and Readable constructors. Also has the following fields:
  - `allowHalfOpen` <boolean> Defaults to `true`. If set to `false`, then the stream will automatically end the writable side when the readable side ends.
  - `readableObjectMode` <boolean> Defaults to `false`. Sets `objectMode` for readable side of the stream. Has no effect if `objectMode` is `true`.
  - `writableObjectMode` <boolean> Defaults to `false`. Sets `objectMode` for writable side of the stream. Has no effect if `objectMode` is `true`.
  - `readableHighWaterMark` <number> Sets `highWaterMark` for the readable side of the stream. Has no effect if `highWaterMark` is provided.
  - `writableHighWaterMark` <number> Sets `highWaterMark` for the writable side of the stream. Has no effect if `highWaterMark` is provided.

```
const { Duplex } = require('stream');

class MyDuplex extends Duplex {
  constructor(options) {
    super(options);
    // ...
  }
}
```

Or, when using pre-ES6 style constructors:

```
const { Duplex } = require('stream');
const util = require('util');

function MyDuplex(options) {
  if (!(this instanceof MyDuplex))
    return new MyDuplex(options);
  Duplex.call(this, options);
}
util.inherits(MyDuplex, Duplex);
```

Or, using the Simplified Constructor approach:

```
const { Duplex } = require('stream');

const myDuplex = new Duplex({
  read(size) {
    // ...
  },
  write(chunk, encoding, callback) {
    // ...
  }
});
```

An Example Duplex Stream

The following illustrates a simple example of a Duplex stream that wraps a hypothetical lower-level source object to which data can be written, and from which data can be read, albeit using an API that is not compatible with Node.js streams. The following illustrates a simple example of a Duplex stream that buffers incoming written data via the Writable interface that is read back out via the Readable interface.

```
const { Duplex } = require('stream');
const kSource = Symbol('source');

class MyDuplex extends Duplex {
  constructor(source, options) {
    super(options);
    this[kSource] = source;
  }

  _write(chunk, encoding, callback) {
    // The underlying source only deals with strings
    if (Buffer.isBuffer(chunk))
      chunk = chunk.toString();
    this[kSource].writeSomeData(chunk);
    callback();
  }

  _read(size) {
    this[kSource].fetchSomeData(size, (data, encoding) => {
      this.push(Buffer.from(data, encoding));
    });
  }
}
```

The most important aspect of a Duplex stream is that the Readable and Writable sides operate independently of one another despite co-existing within a single object instance.

Object Mode Duplex Streams

For Duplex streams, `objectMode` can be set exclusively for either the Readable or Writable side using the `readableObjectMode` and `writableObjectMode` options respectively.

In the following example, for instance, a new Transform stream (which is a type of Duplex stream) is created that has an object mode Writable side that accepts JavaScript numbers that are converted to hexadecimal strings on the Readable side.

```
const { Transform } = require('stream');

// All Transform streams are also Duplex Streams
const myTransform = new Transform({
  writableObjectMode: true,

  transform(chunk, encoding, callback) {
    // Coerce the chunk to a number if necessary
    chunk |= 0;

    // Transform the chunk into something else.
    const data = chunk.toString(16);

    // Push the data onto the readable queue.
    callback(null, '0'.repeat(data.length % 2) + data);
  }
});

myTransform.setEncoding('ascii');
myTransform.on('data', (chunk) => console.log(chunk));

myTransform.write(1);
// Prints: 01
myTransform.write(10);
// Prints: 0a
myTransform.write(100);
// Prints: 64
```

Implementing a Transform Stream

A Transform stream is a Duplex stream where the output is computed in some way from the input. Examples include zlib streams or crypto streams that compress, encrypt, or decrypt data.

There is no requirement that the output be the same size as the input, the same number of chunks, or arrive at the same time. For example, a Hash stream will only ever have a single chunk of output which is provided when the input is ended. A `zlib` stream will produce output that is either much smaller or much larger than its input.

The `stream.Transform` class is extended to implement a Transform stream.

The `stream.Transform` class prototypically inherits from `stream.Duplex` and implements its own versions of the `writable._write()` and `readable._read()` methods. Custom Transform implementations *must* implement the transform._transform() method and *may* also implement the transform._flush() method.

Care must be taken when using Transform streams in that data written to the stream can cause the Writable side of the stream to become paused if the output on the Readable side is not consumed.

new stream.Transform(options)

- `options` <Object> Passed to both Writable and Readable constructors. Also has the following fields:
  - `transform` <Function> Implementation for the stream._transform() method.
  - `flush` <Function> Implementation for the stream._flush() method.

```
const { Transform } = require('stream');

class MyTransform extends Transform {
  constructor(options) {
    super(options);
    // ...
  }
}
```

Or, when using pre-ES6 style constructors:

```
const { Transform } = require('stream');
const util = require('util');

function MyTransform(options) {
  if (!(this instanceof MyTransform))
    return new MyTransform(options);
  Transform.call(this, options);
}
util.inherits(MyTransform, Transform);
```

Or, using the Simplified Constructor approach:

```
const { Transform } = require('stream');

const myTransform = new Transform({
  transform(chunk, encoding, callback) {
    // ...
  }
});
```

Events: 'finish' and 'end'

The `'finish'` and `'end'` events are from the `stream.Writable` and `stream.Readable` classes, respectively. The `'finish'` event is emitted after `stream.end()` is called and all chunks have been processed by `stream._transform()`. The `'end'` event is emitted after all data has been output, which occurs after the callback in `transform._flush()` has been called.

transform._flush(callback)

- `callback` <Function> A callback function (optionally with an error argument and data) to be called when remaining data has been flushed.

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal Readable class methods only.

In some cases, a transform operation may need to emit an additional bit of data at the end of the stream. For example, a `zlib` compression stream will store an amount of internal state used to optimally compress the output. When the stream ends, however, that additional data needs to be flushed so that the compressed data will be complete.

Custom Transform implementations *may* implement the `transform._flush()` method. This will be called when there is no more written data to be consumed, but before the `'end'` event is emitted signaling the end of the Readable stream.

Within the `transform._flush()` implementation, the `readable.push()` method may be called zero or more times, as appropriate. The `callback` function must be called when the flush operation is complete.

The `transform._flush()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

transform._transform(chunk, encoding, callback)

- `chunk` <Buffer> | <string> | <any> The chunk to be transformed. Will **always** be a buffer unless the `decodeStrings` option was set to `false` or the stream is operating in object mode.
- `encoding` <string> If the chunk is a string, then this is the encoding type. If chunk is a buffer, then this is the special value - 'buffer', ignore it in this case.
- `callback` <Function> A callback function (optionally with an error argument and data) to be called after the supplied `chunk` has been processed.

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal Readable class methods only.

All Transform stream implementations must provide a `_transform()` method to accept input and produce output. The `transform._transform()` implementation handles the bytes being written, computes an output, then passes that output off to the readable portion using the `readable.push()` method.

The `transform.push()` method may be called zero or more times to generate output from a single input chunk, depending on how much is to be output as a result of the chunk.

It is possible that no output is generated from any given chunk of input data.

The `callback` function must be called only when the current chunk is completely consumed. The first argument passed to the `callback` must be an `Error` object if an error occurred while processing the input or `null` otherwise. If a second argument is passed to the `callback`, it will be forwarded on to the `readable.push()` method. In other words the following are equivalent:

```
transform.prototype._transform = function(data, encoding, callback) {
  this.push(data);
  callback();
};

transform.prototype._transform = function(data, encoding, callback) {
  callback(null, data);
};
```

The `transform._transform()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

`transform._transform()` is never called in parallel; streams implement a queue mechanism, and to receive the next chunk, `callback` must be called, either synchronously or asynchronously.

Class: stream.PassThrough

The `stream.PassThrough` class is a trivial implementation of a Transform stream that simply passes the input bytes across to the output. Its purpose is primarily for examples and testing, but there are some use cases where `stream.PassThrough` is useful as a building block for novel sorts of streams.

Additional Notes

Compatibility with Older Node.js Versions

In versions of Node.js prior to v0.10, the Readable stream interface was simpler, but also less powerful and less useful.

- Rather than waiting for calls the `stream.read()` method, `'data'` events would begin emitting immediately. Applications that would need to perform some amount of work to decide how to handle data were required to store read data into buffers so the data would not be lost.
- The `stream.pause()` method was advisory, rather than guaranteed. This meant that it was still necessary to be prepared to receive `'data'` events *even when the stream was in a paused state*.

In Node.js v0.10, the Readable class was added. For backwards compatibility with older Node.js programs, Readable streams switch into "flowing mode" when a `'data'` event handler is added, or when the `stream.resume()` method is called. The effect is that, even when not using the new `stream.read()` method and `'readable'` event, it is no longer necessary to worry about losing `'data'` chunks.

While most applications will continue to function normally, this introduces an edge case in the following conditions:

- No `'data'` event listener is added.
- The `stream.resume()` method is never called.
- The stream is not piped to any writable destination.

For example, consider the following code:

```
// WARNING!  BROKEN!
net.createServer((socket) => {

  // we add an 'end' method, but never consume the data
  socket.on('end', () => {
    // It will never get here.
    socket.end('The message was received but was not processed.\n');
  });

}).listen(1337);
```

In versions of Node.js prior to v0.10, the incoming message data would be simply discarded. However, in Node.js v0.10 and beyond, the socket remains paused forever.

The workaround in this situation is to call the `stream.resume()` method to begin the flow of data:

```
// Workaround
net.createServer((socket) => {

  socket.on('end', () => {
    socket.end('The message was received but was not processed.\n');
  });

  // start the flow of data, discarding it.
  socket.resume();

}).listen(1337);
```

In addition to new Readable streams switching into flowing mode, pre-v0.10 style streams can be wrapped in a Readable class using the `readable.wrap()` method.

readable.read(0)

There are some cases where it is necessary to trigger a refresh of the underlying readable stream mechanisms, without actually consuming any data. In such cases, it is possible to call `readable.read(0)`, which will always return `null`.

If the internal read buffer is below the `highWaterMark`, and the stream is not currently reading, then calling `stream.read(0)` will trigger a low-level `stream._read()` call.

While most applications will almost never need to do this, there are situations within Node.js where this is done, particularly in the Readable stream class internals.

readable.push('')

Use of `readable.push('')` is not recommended.

Pushing a zero-byte string, `Buffer` or `Uint8Array` to a stream that is not in object mode has an interesting side effect. Because it *is* a call to `readable.push()`, the call will end the reading process. However, because the argument is an empty string, no data is added to the readable buffer so there is nothing for a user to consume.

`highWaterMark` discrepancy after calling `readable.setEncoding()`

The use of `readable.setEncoding()` will change the behavior of how the `highWaterMark` operates in non-object mode.

Typically, the size of the current buffer is measured against the `highWaterMark` in *bytes*. However, after `setEncoding()` is called, the comparison function will begin to measure the buffer's size in *characters*.

This is not a problem in common cases with `latin1` or `ascii`. But it is advised to be mindful about this behavior when working with strings that could contain multi-byte characters.

String Decoder

Stability: 2 - Stable

The `string_decoder` module provides an API for decoding `Buffer` objects into strings in a manner that preserves encoded multi-byte UTF-8 and UTF-16 characters. It can be accessed using:

```
const { StringDecoder } = require('string_decoder');
```

The following example shows the basic use of the `StringDecoder` class.

```
const { StringDecoder } = require('string_decoder');
const decoder = new StringDecoder('utf8');

const cent = Buffer.from([0xC2, 0xA2]);
console.log(decoder.write(cent));

const euro = Buffer.from([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

When a `Buffer` instance is written to the `StringDecoder` instance, an internal buffer is used to ensure that the decoded string does not contain any incomplete multibyte characters. These are held in the buffer until the next call to `stringDecoder.write()` or until `stringDecoder.end()` is called.

In the following example, the three UTF-8 encoded bytes of the European Euro symbol (`€`) are written over three separate operations:

```
const { StringDecoder } = require('string_decoder');
const decoder = new StringDecoder('utf8');

decoder.write(Buffer.from([0xE2]));
decoder.write(Buffer.from([0x82]));
console.log(decoder.end(Buffer.from([0xAC])));
```

Class: new StringDecoder([encoding])

Added in: v0.1.99

- `encoding` <string> The character encoding the `StringDecoder` will use. Defaults to `'utf8'`.

Creates a new `StringDecoder` instance.

stringDecoder.end(buffer)

Added in: v0.9.3

- `buffer` <Buffer> A `Buffer` containing the bytes to decode.

Returns any remaining input stored in the internal buffer as a string. Bytes representing incomplete UTF-8 and UTF-16 characters will be

replaced with substitution characters appropriate for the character encoding.

If the `buffer` argument is provided, one final call to `stringDecoder.write()` is performed before returning the remaining input.

stringDecoder.write(buffer)

▸ History

- `buffer` <Buffer> A `Buffer` containing the bytes to decode.

Returns a decoded string, ensuring that any incomplete multibyte characters at the end of the `Buffer` are omitted from the returned string and stored in an internal buffer for the next call to `stringDecoder.write()` or `stringDecoder.end()`.

Timers

Stability: 2 - Stable

The `timer` module exposes a global API for scheduling functions to be called at some future period of time. Because the timer functions are globals, there is no need to call `require('timers')` to use the API.

The timer functions within Node.js implement a similar API as the timers API provided by Web Browsers but use a different internal implementation that is built around the Node.js Event Loop.

Class: Immediate

This object is created internally and is returned from `setImmediate()`. It can be passed to `clearImmediate()` in order to cancel the scheduled actions.

By default, when an immediate is scheduled, the Node.js event loop will continue running as long as the immediate is active. The `Immediate` object returned by `setImmediate()` exports both `immediate.ref()` and `immediate.unref()` functions that can be used to control this default behavior.

immediate.ref()

Added in: v9.7.0

When called, requests that the Node.js event loop *not* exit so long as the `Immediate` is active. Calling `immediate.ref()` multiple times will have no effect.

By default, all `Immediate` objects are "ref'd", making it normally unnecessary to call `immediate.ref()` unless `immediate.unref()` had been called previously.

Returns a reference to the `Immediate`.

immediate.unref()

Added in: v9.7.0

When called, the active `Immediate` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `Immediate` object's callback is invoked. Calling `immediate.unref()` multiple times will have no effect.

Returns a reference to the `Immediate`.

Class: Timeout

This object is created internally and is returned from `setTimeout()` and `setInterval()`. It can be passed to `clearTimeout()` or `clearInterval()` (respectively) in order to cancel the scheduled actions.

By default, when a timer is scheduled using either `setTimeout()` or `setInterval()`, the Node.js event loop will continue running as long as the

timer is active. Each of the `Timeout` objects returned by these functions export both `timeout.ref()` and `timeout.unref()` functions that can be used to control this default behavior.

timeout.ref()

Added in: v0.9.1

When called, requests that the Node.js event loop *not* exit so long as the `Timeout` is active. Calling `timeout.ref()` multiple times will have no effect.

By default, all `Timeout` objects are "ref'd", making it normally unnecessary to call `timeout.ref()` unless `timeout.unref()` had been called previously.

Returns a reference to the `Timeout`.

timeout.unref()

Added in: v0.9.1

When called, the active `Timeout` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `Timeout` object's callback is invoked. Calling `timeout.unref()` multiple times will have no effect.

Calling `timeout.unref()` creates an internal timer that will wake the Node.js event loop. Creating too many of these can adversely impact performance of the Node.js application.

Returns a reference to the `Timeout`.

Scheduling Timers

A timer in Node.js is an internal construct that calls a given function after a certain period of time. When a timer's function is called varies depending on which method was used to create the timer and what other work the Node.js event loop is doing.

setImmediate(callback[, ...args])

Added in: v0.9.1

- `callback` <Function> The function to call at the end of this turn of the Node.js Event Loop
- `...args` <any> Optional arguments to pass when the `callback` is called.

Schedules the "immediate" execution of the `callback` after I/O events' callbacks. Returns an `Immediate` for use with `clearImmediate()`.

When multiple calls to `setImmediate()` are made, the `callback` functions are queued for execution in the order in which they are created. The entire callback queue is processed every event loop iteration. If an immediate timer is queued from inside an executing callback, that timer will not be triggered until the next event loop iteration.

If `callback` is not a function, a `TypeError` will be thrown.

This method has a custom variant for promises that is available using `util.promisify()`:

```
const util = require('util');
const setImmediatePromise = util.promisify(setImmediate);

setImmediatePromise('foobar').then((value) => {
  // value === 'foobar' (passing values is optional)
  // This is executed after all I/O callbacks.
});

// or with async function
async function timerExample() {
  console.log('Before I/O callbacks');
  await setImmediatePromise();
  console.log('After I/O callbacks');
}
timerExample();
```

setInterval(callback, delay[, ...args])

Added in: v0.0.1

- `callback` <Function> The function to call when the timer elapses.
- `delay` <number> The number of milliseconds to wait before calling the `callback`.
- `...args` <any> Optional arguments to pass when the `callback` is called.

Schedules repeated execution of `callback` every `delay` milliseconds. Returns a `Timeout` for use with clearInterval().

When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`.

If `callback` is not a function, a TypeError will be thrown.

setTimeout(callback, delay[, ...args])

Added in: v0.0.1

- `callback` <Function> The function to call when the timer elapses.
- `delay` <number> The number of milliseconds to wait before calling the `callback`.
- `...args` <any> Optional arguments to pass when the `callback` is called.

Schedules execution of a one-time `callback` after `delay` milliseconds. Returns a `Timeout` for use with clearTimeout().

The `callback` will likely not be invoked in precisely `delay` milliseconds. Node.js makes no guarantees about the exact timing of when callbacks will fire, nor of their ordering. The callback will be called as close as possible to the time specified.

When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`.

If `callback` is not a function, a TypeError will be thrown.

This method has a custom variant for promises that is available using util.promisify():

```
const util = require('util');
const setTimeoutPromise = util.promisify(setTimeout);

setTimeoutPromise(40, 'foobar').then((value) => {
  // value === 'foobar' (passing values is optional)
  // This is executed after about 40 milliseconds.
});
```

Cancelling Timers

The `setImmediate()`, `setInterval()`, and `setTimeout()` methods each return objects that represent the scheduled timers. These can be used to cancel the timer and prevent it from triggering.

It is not possible to cancel timers that were created using the promisified variants of `setImmediate()`, `setTimeout()`.

clearImmediate(immediate)

Added in: v0.9.1

- `immediate` <Immediate> An `Immediate` object as returned by `setImmediate()`.

Cancels an `Immediate` object created by `setImmediate()`.

clearInterval(timeout)

Added in: v0.0.1

- `timeout` <Timeout> A `Timeout` object as returned by `setInterval()`.

Cancels a `Timeout` object created by `setInterval()`.

clearTimeout(timeout)

Added in: v0.0.1

- `timeout` <Timeout> A `Timeout` object as returned by `setTimeout()`.

Cancels a `Timeout` object created by `setTimeout()`.

TLS (SSL)

Stability: 2 - Stable

The `tls` module provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols that is built on top of OpenSSL. The module can be accessed using:

```
const tls = require('tls');
```

TLS/SSL Concepts

The TLS/SSL is a public/private key infrastructure (PKI). For most common cases, each client and server must have a *private key*.

Private keys can be generated in multiple ways. The example below illustrates use of the OpenSSL command-line interface to generate a 2048-bit RSA private key:

```
openssl genrsa -out ryans-key.pem 2048
```

With TLS/SSL, all servers (and some clients) must have a *certificate*. Certificates are *public keys* that correspond to a private key, and that are digitally signed either by a Certificate Authority or by the owner of the private key (such certificates are referred to as "self-signed"). The first step to obtaining a certificate is to create a *Certificate Signing Request* (CSR) file.

The OpenSSL command-line interface can be used to generate a CSR for a private key:

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

Once the CSR file is generated, it can either be sent to a Certificate Authority for signing or used to generate a self-signed certificate.

Creating a self-signed certificate using the OpenSSL command-line interface is illustrated in the example below:

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem
```

Once the certificate is generated, it can be used to generate a `.pfx` or `.p12` file:

```
openssl pkcs12 -export -in ryans-cert.pem -inkey ryans-key.pem \
      -certfile ca-cert.pem -out ryans.pfx
```

Where:

- `in` : is the signed certificate
- `inkey` : is the associated private key
- `certfile` : is a concatenation of all Certificate Authority (CA) certs into a single file, e.g. `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

Perfect Forward Secrecy

The term " Forward Secrecy " or "Perfect Forward Secrecy" describes a feature of key-agreement (i.e., key-exchange) methods. That is, the server and client keys are used to negotiate new temporary keys that are used specifically and only for the current communication session. Practically, this means that even if the server's private key is compromised, communication can only be decrypted by eavesdroppers if the attacker manages to obtain the key-pair specifically generated for the session.

Perfect Forward Secrecy is achieved by randomly generating a key pair for key-agreement on every TLS/SSL handshake (in contrast to using the same key for all sessions). Methods implementing this technique are called "ephemeral".

Currently two methods are commonly used to achieve Perfect Forward Secrecy (note the character "E" appended to the traditional abbreviations):

- DHE - An ephemeral version of the Diffie Hellman key-agreement protocol.
- ECDHE - An ephemeral version of the Elliptic Curve Diffie Hellman key-agreement protocol.

Ephemeral methods may have some performance drawbacks, because key generation is expensive.

To use Perfect Forward Secrecy using `DHE` with the `tls` module, it is required to generate Diffie-Hellman parameters and specify them with the `dhparam` option to `tls.createSecureContext()` . The following illustrates the use of the OpenSSL command-line interface to generate such parameters:

```
openssl dhparam -outform PEM -out dhparam.pem 2048
```

If using Perfect Forward Secrecy using `ECDHE`, Diffie-Hellman parameters are not required and a default ECDHE curve will be used. The `ecdhCurve` property can be used when creating a TLS Server to specify the list of names of supported curves to use, see `tls.createServer()` for more info.

ALPN, NPN, and SNI

ALPN (Application-Layer Protocol Negotiation Extension), NPN (Next Protocol Negotiation) and, SNI (Server Name Indication) are TLS handshake extensions:

- ALPN/NPN - Allows the use of one TLS server for multiple protocols (HTTP, SPDY, HTTP/2)
- SNI - Allows the use of one TLS server for multiple hostnames with different SSL certificates.

Use of ALPN is recommended over NPN. The NPN extension has never been formally defined or documented and generally not recommended for use.

Client-initiated renegotiation attack mitigation

The TLS protocol allows clients to renegotiate certain aspects of the TLS session. Unfortunately, session renegotiation requires a disproportionate amount of server-side resources, making it a potential vector for denial-of-service attacks.

To mitigate the risk, renegotiation is limited to three times every ten minutes. An `'error'` event is emitted on the `tls.TLSSocket` instance when this threshold is exceeded. The limits are configurable:

- `tls.CLIENT_RENEG_LIMIT` <number> Specifies the number of renegotiation requests. Defaults to `3`.
- `tls.CLIENT_RENEG_WINDOW` <number> Specifies the time renegotiation window in seconds. Defaults to `600` (10 minutes).

The default renegotiation limits should not be modified without a full understanding of the implications and risks.

To test the renegotiation limits on a server, connect to it using the OpenSSL command-line client (`openssl s_client -connect address:port`) then input `R<CR>` (i.e., the letter `R` followed by a carriage return) multiple times.

Modifying the Default TLS Cipher suite

Node.js is built with a default suite of enabled and disabled TLS ciphers. Currently, the default cipher suite is:

```
ECDHE-RSA-AES128-GCM-SHA256:
ECDHE-ECDSA-AES128-GCM-SHA256:
ECDHE-RSA-AES256-GCM-SHA384:
ECDHE-ECDSA-AES256-GCM-SHA384:
DHE-RSA-AES128-GCM-SHA256:
ECDHE-RSA-AES128-SHA256:
DHE-RSA-AES128-SHA256:
ECDHE-RSA-AES256-SHA384:
DHE-RSA-AES256-SHA384:
ECDHE-RSA-AES256-SHA256:
DHE-RSA-AES256-SHA256:
HIGH:
!aNULL:
!eNULL:
!EXPORT:
!DES:
!RC4:
!MD5:
!PSK:
!SRP:
!CAMELLIA
```

This default can be replaced entirely using the `--tls-cipher-list` command line switch. For instance, the following makes `ECDHE-RSA-AES128-GCM-SHA256:!RC4` the default TLS cipher suite:

```
node --tls-cipher-list="ECDHE-RSA-AES128-GCM-SHA256:!RC4"
```

The default can also be replaced on a per client or server basis using the `ciphers` option from `tls.createSecureContext()`, which is also available in `tls.createServer()`, `tls.connect()`, and when creating new `tls.TLSSocket`s.

Consult OpenSSL cipher list format documentation for details on the format.

The default cipher suite included within Node.js has been carefully selected to reflect current security best practices and risk mitigation. Changing the default cipher suite can have a significant impact on the security of an application. The `--tls-cipher-list` switch and `ciphers` option should by used only if absolutely necessary.

The default cipher suite prefers GCM ciphers for Chrome's 'modern cryptography' setting and also prefers ECDHE and DHE ciphers for Perfect Forward Secrecy, while offering *some* backward compatibility.

128 bit AES is preferred over 192 and 256 bit AES in light of specific attacks affecting larger AES key sizes.

Old clients that rely on insecure and deprecated RC4 or DES-based ciphers (like Internet Explorer 6) cannot complete the handshaking process with the default configuration. If these clients *must* be supported, the TLS recommendations may offer a compatible cipher suite. For more details on the format, see the OpenSSL cipher list format documentation.

Class: tls.Server

Added in: v0.3.2

The `tls.Server` class is a subclass of `net.Server` that accepts encrypted connections using TLS or SSL.

Event: 'newSession'

Added in: v0.9.2

The `'newSession'` event is emitted upon creation of a new TLS session. This may be used to store sessions in external storage. The listener callback is passed three arguments when called:

- `sessionId` - The TLS session identifier
- `sessionData` - The TLS session data
- `callback` <Function> A callback function taking no arguments that must be invoked in order for data to be sent or received over the secure connection.

Listening for this event will have an effect only on connections established after the addition of the event listener.

Event: 'OCSPRequest'

Added in: v0.11.13

The `'OCSPRequest'` event is emitted when the client sends a certificate status request. The listener callback is passed three arguments when called:

- `certificate` <Buffer> The server certificate
- `issuer` <Buffer> The issuer's certificate
- `callback` <Function> A callback function that must be invoked to provide the results of the OCSP request.

The server's current certificate can be parsed to obtain the OCSP URL and certificate ID; after obtaining an OCSP response, `callback(null, resp)` is then invoked, where `resp` is a `Buffer` instance containing the OCSP response. Both `certificate` and `issuer` are `Buffer` DER-representations of the primary and issuer's certificates. These can be used to obtain the OCSP certificate ID and OCSP endpoint URL.

Alternatively, `callback(null, null)` may be called, indicating that there was no OCSP response.

Calling `callback(err)` will result in a `socket.destroy(err)` call.

The typical flow of an OCSP Request is as follows:

1. Client connects to the server and sends an `'OCSPRequest'` (via the status info extension in ClientHello).
2. Server receives the request and emits the `'OCSPRequest'` event, calling the listener if registered.
3. Server extracts the OCSP URL from either the `certificate` or `issuer` and performs an OCSP request to the CA.
4. Server receives `OCSPResponse` from the CA and sends it back to the client via the `callback` argument
5. Client validates the response and either destroys the socket or performs a handshake.

The `issuer` can be `null` if the certificate is either self-signed or the issuer is not in the root certificates list. (An issuer may be provided via the `ca` option when establishing the TLS connection.)

Listening for this event will have an effect only on connections established after the addition of the event listener.

An npm module like asn1.js may be used to parse the certificates.

Event: 'resumeSession'

Added in: v0.9.2

The `'resumeSession'` event is emitted when the client requests to resume a previous TLS session. The listener callback is passed two arguments when called:

- `sessionId` - The TLS/SSL session identifier
- `callback` <Function> A callback function to be called when the prior session has been recovered.

When called, the event listener may perform a lookup in external storage using the given `sessionId` and invoke `callback(null, sessionData)` once finished. If the session cannot be resumed (i.e., doesn't exist in storage) the callback may be invoked as `callback(null, null)`. Calling `callback(err)`

will terminate the incoming connection and destroy the socket.

Listening for this event will have an effect only on connections established after the addition of the event listener.

The following illustrates resuming a TLS session:

```
const tlsSessionStore = {};
server.on('newSession', (id, data, cb) => {
  tlsSessionStore[id.toString('hex')] = data;
  cb();
});
server.on('resumeSession', (id, cb) => {
  cb(null, tlsSessionStore[id.toString('hex')] || null);
});
```

Event: 'secureConnection'

Added in: v0.3.2

The `'secureConnection'` event is emitted after the handshaking process for a new connection has successfully completed. The listener callback is passed a single argument when called:

- `tlsSocket` <tls.TLSSocket> The established TLS socket.

The `tlsSocket.authorized` property is a `boolean` indicating whether the client has been verified by one of the supplied Certificate Authorities for the server. If `tlsSocket.authorized` is `false`, then `socket.authorizationError` is set to describe how authorization failed. Note that depending on the settings of the TLS server, unauthorized connections may still be accepted.

The `tlsSocket.npnProtocol` and `tlsSocket.alpnProtocol` properties are strings that contain the selected NPN and ALPN protocols, respectively. When both NPN and ALPN extensions are received, ALPN takes precedence over NPN and the next protocol is selected by ALPN.

When ALPN has no selected protocol, `tlsSocket.alpnProtocol` returns `false`.

The `tlsSocket.servername` property is a string containing the server name requested via SNI.

Event: 'tlsClientError'

Added in: v6.0.0

The `'tlsClientError'` event is emitted when an error occurs before a secure connection is established. The listener callback is passed two arguments when called:

- `exception` <Error> The `Error` object describing the error
- `tlsSocket` <tls.TLSSocket> The `tls.TLSSocket` instance from which the error originated.

server.addContext(hostname, context)

Added in: v0.5.3

- `hostname` <string> A SNI hostname or wildcard (e.g. `'*'`)
- `context` <Object> An object containing any of the possible properties from the tls.createSecureContext() `options` arguments (e.g. `key`, `cert`, `ca`, etc).

The `server.addContext()` method adds a secure context that will be used if the client request's SNI hostname matches the supplied `hostname` (or wildcard).

server.address()

Added in: v0.6.0

Returns the bound address, the address family name, and port of the server as reported by the operating system. See net.Server.address() for more information.

server.close([callback])

Added in: v0.3.2

- `callback` <Function> An optional listener callback that will be registered to listen for the server instance's `'close'` event.

The `server.close()` method stops the server from accepting new connections.

This function operates asynchronously. The `'close'` event will be emitted when the server has no more open connections.

server.connections

Added in: v0.3.2 Deprecated since: v0.9.7

Stability: 0 - Deprecated: Use server.getConnections() instead.

Returns the current number of concurrent connections on the server.

server.getTicketKeys()

Added in: v3.0.0

Returns a `Buffer` instance holding the keys currently used for encryption/decryption of the TLS Session Tickets

server.listen()

Starts the server listening for encrypted connections. This method is identical to `server.listen()` from `net.Server` .

server.setTicketKeys(keys)

Added in: v3.0.0

- `keys` <Buffer> The keys used for encryption/decryption of the TLS Session Tickets .

Updates the keys for encryption/decryption of the TLS Session Tickets .

The key's `Buffer` should be 48 bytes long. See `ticketKeys` option in tls.createServer for more information on how it is used.

Changes to the ticket keys are effective only for future server connections. Existing or currently pending server connections will use the previous keys.

Class: tls.TLSSocket

Added in: v0.11.4

The `tls.TLSSocket` is a subclass of `net.Socket` that performs transparent encryption of written data and all required TLS negotiation.

Instances of `tls.TLSSocket` implement the duplex Stream interface.

Methods that return TLS connection metadata (e.g. `tls.TLSSocket.getPeerCertificate()` will only return data while the connection is open.

new tls.TLSSocket(socket[, options])

▸ History

- `socket` <net.Socket> | <stream.Duplex> On the server side, any `Duplex` stream. On the client side, any instance of `net.Socket` (for generic `Duplex` stream support on the client side, `tls.connect()` must be used).
- `options` <Object>
  - `isServer` : The SSL/TLS protocol is asymmetrical, TLSSockets must know if they are to behave as a server or a client. If `true` the TLS socket will be instantiated as a server. Defaults to `false` .
  - `server` <net.Server> An optional `net.Server` instance.
  - `requestCert` : Whether to authenticate the remote peer by requesting a certificate. Clients always request a server certificate.

Servers (`isServer` is true) may optionally set `requestCert` to true to request a client certificate.

  - `rejectUnauthorized` : Optional, see tls.createServer()
  - `NPNProtocols` : Optional, see tls.createServer()
  - `ALPNProtocols` : Optional, see tls.createServer()
  - `SNICallback` : Optional, see tls.createServer()
  - `session` <Buffer> An optional `Buffer` instance containing a TLS session.
  - `requestOCSP` <boolean> If `true` , specifies that the OCSP status request extension will be added to the client hello and an `'OCSPResponse'` event will be emitted on the socket before establishing a secure communication
  - `secureContext` : Optional TLS context object created with tls.createSecureContext() . If a `secureContext` is *not* provided, one will be created by passing the entire `options` object to `tls.createSecureContext()` .
  - ...: Optional tls.createSecureContext() options that are used if the `secureContext` option is missing, otherwise they are ignored.

Construct a new `tls.TLSSocket` object from an existing TCP socket.

Event: 'OCSPResponse'

Added in: v0.11.13

The `'OCSPResponse'` event is emitted if the `requestOCSP` option was set when the `tls.TLSSocket` was created and an OCSP response has been received. The listener callback is passed a single argument when called:

- `response` <Buffer> The server's OCSP response

Typically, the `response` is a digitally signed object from the server's CA that contains information about server's certificate revocation status.

Event: 'secureConnect'

Added in: v0.11.4

The `'secureConnect'` event is emitted after the handshaking process for a new connection has successfully completed. The listener callback will be called regardless of whether or not the server's certificate has been authorized. It is the client's responsibility to check the `tlsSocket.authorized` property to determine if the server certificate was signed by one of the specified CAs. If `tlsSocket.authorized === false` , then the error can be found by examining the `tlsSocket.authorizationError` property. If either ALPN or NPN was used, the `tlsSocket.alpnProtocol` or `tlsSocket.npnProtocol` properties can be checked to determine the negotiated protocol.

tlsSocket.address()

Added in: v0.11.4

Returns the bound address, the address family name, and port of the underlying socket as reported by the operating system. Returns an object with three properties, e.g. `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

tlsSocket.authorizationError

Added in: v0.11.4

Returns the reason why the peer's certificate was not been verified. This property is set only when `tlsSocket.authorized === false` .

tlsSocket.authorized

Added in: v0.11.4

Returns `true` if the peer certificate was signed by one of the CAs specified when creating the `tls.TLSSocket` instance, otherwise `false` .

tlsSocket.disableRenegotiation()

Added in: v8.4.0

Disables TLS renegotiation for this `TLSSocket` instance. Once called, attempts to renegotiate will trigger an `'error'` event on the `TLSSocket` .

tlsSocket.encrypted

Added in: v0.11.4

Always returns `true`. This may be used to distinguish TLS sockets from regular `net.Socket` instances.

tlsSocket.getCipher()

Added in: v0.11.4

Returns an object representing the cipher name. The `version` key is a legacy field which always contains the value `'TLSv1/SSLv3'`.

For example: `{ name: 'AES256-SHA', version: 'TLSv1/SSLv3' }`

See `SSL_CIPHER_get_name()` in https://www.openssl.org/docs/man1.0.2/ssl/SSL_CIPHER_get_name.html for more information.

tlsSocket.getEphemeralKeyInfo()

Added in: v5.0.0

Returns an object representing the type, name, and size of parameter of an ephemeral key exchange in Perfect Forward Secrecy on a client connection. It returns an empty object when the key exchange is not ephemeral. As this is only supported on a client socket; `null` is returned if called on a server socket. The supported types are `'DH'` and `'ECDH'`. The `name` property is available only when type is 'ECDH'.

For Example: `{ type: 'ECDH', name: 'prime256v1', size: 256 }`

tlsSocket.getFinished()

Added in: v9.9.0

- Returns: <Buffer> | <undefined> The latest `Finished` message that has been sent to the socket as part of a SSL/TLS handshake, or `undefined` if no `Finished` message has been sent yet.

As the `Finished` messages are message digests of the complete handshake (with a total of 192 bits for TLS 1.0 and more for SSL 3.0), they can be used for external authentication procedures when the authentication provided by SSL/TLS is not desired or is not enough.

Corresponds to the `SSL_get_finished` routine in OpenSSL and may be used to implement the `tls-unique` channel binding from RFC 5929.

tlsSocket.getPeerCertificate([detailed])

Added in: v0.11.4

- `detailed` <boolean> Include the full certificate chain if `true`, otherwise include just the peer's certificate.

Returns an object representing the peer's certificate. The returned object has some properties corresponding to the fields of the certificate.

If the full certificate chain was requested, each certificate will include a `issuerCertificate` property containing an object representing its issuer's certificate.

For example:

```
{ subject:
  { C: 'UK',
    ST: 'Acknack Ltd',
    L: 'Rhys Jones',
    O: 'node.js',
    OU: 'Test TLS Certificate',
    CN: 'localhost' },
  issuer:
  { C: 'UK',
    ST: 'Acknack Ltd',
    L: 'Rhys Jones',
    O: 'node.js',
    OU: 'Test TLS Certificate',
    CN: 'localhost' },
  issuerCertificate:
  { ... another certificate, possibly with a .issuerCertificate ... },
  raw: < RAW DER buffer >,
  pubkey: < RAW DER buffer >,
  valid_from: 'Nov 11 09:52:22 2009 GMT',
  valid_to: 'Nov 6 09:52:22 2029 GMT',
  fingerprint: '2A:7A:C2:DD:E5:F9:CC:53:72:35:99:7A:02:5A:71:38:52:EC:8A:DF',
  fingerprint256: '2A:7A:C2:DD:E5:F9:CC:53:72:35:99:7A:02:5A:71:38:52:EC:8A:DF:00:11:22:33:44:55:66:77:88:99:AA:BB',
  serialNumber: 'B9B0D332A1AA5635' }
```

If the peer does not provide a certificate, an empty object will be returned.

tlsSocket.getPeerFinished()

Added in: v9.9.0

- Returns: <Buffer> | <undefined> The latest `Finished` message that is expected or has actually been received from the socket as part of a SSL/TLS handshake, or `undefined` if there is no `Finished` message so far.

As the `Finished` messages are message digests of the complete handshake (with a total of 192 bits for TLS 1.0 and more for SSL 3.0), they can be used for external authentication procedures when the authentication provided by SSL/TLS is not desired or is not enough.

Corresponds to the `SSL_get_peer_finished` routine in OpenSSL and may be used to implement the `tls-unique` channel binding from RFC 5929.

tlsSocket.getProtocol()

Added in: v5.7.0

Returns a string containing the negotiated SSL/TLS protocol version of the current connection. The value `'unknown'` will be returned for connected sockets that have not completed the handshaking process. The value `null` will be returned for server sockets or disconnected client sockets.

Example responses include:

- `SSLv3`
- `TLSv1`
- `TLSv1.1`
- `TLSv1.2`
- `unknown`

See https://www.openssl.org/docs/man1.0.2/ssl/SSL_get_version.html for more information.

tlsSocket.getSession()

Added in: v0.11.4

Returns the ASN.1 encoded TLS session or `undefined` if no session was negotiated. Can be used to speed up handshake establishment when reconnecting to the server.

tlsSocket.getTLSTicket()

Added in: v0.11.4

Returns the TLS session ticket or `undefined` if no session was negotiated.

This only works with client TLS sockets. Useful only for debugging, for session reuse provide `session` option to `tls.connect()`.

tlsSocket.localAddress

Added in: v0.11.4

Returns the string representation of the local IP address.

tlsSocket.localPort

Added in: v0.11.4

Returns the numeric representation of the local port.

tlsSocket.remoteAddress

Added in: v0.11.4

Returns the string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`.

tlsSocket.remoteFamily

Added in: v0.11.4

Returns the string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

tlsSocket.remotePort

Added in: v0.11.4

Returns the numeric representation of the remote port. For example, `443`.

tlsSocket.renegotiate(options, callback)

Added in: v0.11.8

- `options` <Object>
  - `rejectUnauthorized` <boolean> If not `false`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. Defaults to `true`.
  - `requestCert`
- `callback` <Function> A function that will be called when the renegotiation request has been completed.

The `tlsSocket.renegotiate()` method initiates a TLS renegotiation process. Upon completion, the `callback` function will be passed a single argument that is either an `Error` (if the request failed) or `null`.

This method can be used to request a peer's certificate after the secure connection has been established.

When running as the server, the socket will be destroyed with an error after `handshakeTimeout` timeout.

tlsSocket.setMaxSendFragment(size)

Added in: v0.11.11

- `size` <number> The maximum TLS fragment size. Defaults to `16384`. The maximum value is `16384`.

The `tlsSocket.setMaxSendFragment()` method sets the maximum TLS fragment size. Returns `true` if setting the limit succeeded; `false` otherwise.

Smaller fragment sizes decrease the buffering latency on the client: larger fragments are buffered by the TLS layer until the entire fragment is received and its integrity is verified; large fragments can span multiple roundtrips and their processing can be delayed due to packet loss or reordering. However, smaller fragments add extra TLS framing bytes and CPU overhead, which may decrease overall server throughput.

tls.checkServerIdentity(host, cert)

Added in: v0.8.4

- `host` <string> The hostname to verify the certificate against
- `cert` <Object> An object representing the peer's certificate. The returned object has some properties corresponding to the fields of the certificate.

Verifies the certificate `cert` is issued to host `host`.

Returns <Error> object, populating it with the reason, host, and cert on failure. On success, returns <undefined>.

This function can be overwritten by providing alternative function as part of the `options.checkServerIdentity` option passed to `tls.connect()`. The overwriting function can call `tls.checkServerIdentity()` of course, to augment the checks done with additional verification.

This function is only called if the certificate passed all other checks, such as being issued by trusted CA (`options.ca`).

The cert object contains the parsed certificate and will have a structure similar to:

```
{ subject:
   { OU: [ 'Domain Control Validated', 'PositiveSSL Wildcard' ],
     CN: '*.nodejs.org' },
  issuer:
   { C: 'GB',
     ST: 'Greater Manchester',
     L: 'Salford',
     O: 'COMODO CA Limited',
     CN: 'COMODO RSA Domain Validation Secure Server CA' },
  subjectaltname: 'DNS:*.nodejs.org, DNS:nodejs.org',
  infoAccess:
   { 'CA Issuers - URI':
      [ 'http://crt.comodoca.com/COMODORSADomainValidationSecureServerCA.crt' ],
     'OCSP - URI': [ 'http://ocsp.comodoca.com' ] },
  modulus: 'B56CE45CB740B09A13F64AC543B712FF9EE8E4C284B542A1708A27E82A8D151CA178153E12E6DDA15BF70FFD96CB8A88618641BDFCCA03
  exponent: '0x10001',
  pubkey: <Buffer ... >,
  valid_from: 'Aug 14 00:00:00 2017 GMT',
  valid_to: 'Nov 20 23:59:59 2019 GMT',
  fingerprint: '01:02:59:D9:C3:D2:0D:08:F7:82:4E:44:A4:B4:53:C5:E2:3A:87:4D',
  fingerprint256: '69:AE:1A:6A:D4:3D:C6:C1:1B:EA:C6:23:DE:BA:2A:14:62:62:93:5C:7A:EA:06:41:9B:0B:BC:87:CE:48:4E:02',
  ext_key_usage: [ '1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2' ],
  serialNumber: '66593D57F20CBC573E433381B5FEC280',
  raw: <Buffer ... > }
```

tls.connect(options[, callback])

▸ History

- `options` <Object>
  - `host` <string> Host the client should connect to, defaults to 'localhost'.
  - `port` <number> Port the client should connect to.
  - `path` <string> Creates unix socket connection to path. If this option is specified, `host` and `port` are ignored.
  - `socket` <stream.Duplex> Establish secure connection on a given socket rather than creating a new socket. Typically, this is an

instance of net.Socket, but any `Duplex` stream is allowed. If this option is specified, `path`, `host` and `port` are ignored, except for certificate validation. Usually, a socket is already connected when passed to `tls.connect()`, but it can be connected later. Note that connection/disconnection/destruction of `socket` is the user's responsibility, calling `tls.connect()` will not cause `net.connect()` to be called.

  - `rejectUnauthorized` <boolean> If not `false`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. Defaults to `true`.
  - `NPNProtocols` <string[]> | <Buffer[]> | <Uint8Array[]> | <Buffer> | <Uint8Array> An array of strings, `Buffer`s or `Uint8Array`s, or a single `Buffer` or `Uint8Array` containing supported NPN protocols. `Buffer`s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g `['hello', 'world']`.
  - `ALPNProtocols`: <string[]> | <Buffer[]> | <Uint8Array[]> | <Buffer> | <Uint8Array> An array of strings, `Buffer`s or `Uint8Array`s, or a single `Buffer` or `Uint8Array` containing the supported ALPN protocols. `Buffer`s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g `['hello', 'world']`.
  - `servername`: <string> Server name for the SNI (Server Name Indication) TLS extension.
  - `checkServerIdentity(servername, cert)` <Function> A callback function to be used (instead of the builtin `tls.checkServerIdentity()` function) when checking the server's hostname (or the provided `servername` when explicitly set) against the certificate. This should return an <Error> if verification fails. The method should return `undefined` if the `servername` and `cert` are verified.
  - `session` <Buffer> A `Buffer` instance, containing TLS session.
  - `minDHSize` <number> Minimum size of the DH parameter in bits to accept a TLS connection. When a server offers a DH parameter with a size less than `minDHSize`, the TLS connection is destroyed and an error is thrown. Defaults to `1024`.
  - `secureContext`: Optional TLS context object created with tls.createSecureContext(). If a `secureContext` is *not* provided, one will be created by passing the entire `options` object to `tls.createSecureContext()`.
  - `lookup`: <Function> Custom lookup function. Defaults to dns.lookup().
  - ...: Optional tls.createSecureContext() options that are used if the `secureContext` option is missing, otherwise they are ignored.

- `callback` <Function>

The `callback` function, if specified, will be added as a listener for the 'secureConnect' event.

`tls.connect()` returns a tls.TLSSocket object.

The following implements a simple "echo server" example:

```js
const tls = require('tls');
const fs = require('fs');

const options = {
  // Necessary only if using the client certificate authentication
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),

  // Necessary only if the server uses the self-signed certificate
  ca: [ fs.readFileSync('server-cert.pem') ]
};

const socket = tls.connect(8000, options, () => {
  console.log('client connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', (data) => {
  console.log(data);
});
socket.on('end', () => {
  server.close();
});
```

Or

```js
const tls = require('tls');
const fs = require('fs');

const options = {
  pfx: fs.readFileSync('client.pfx')
};

const socket = tls.connect(8000, options, () => {
  console.log('client connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', (data) => {
  console.log(data);
});
socket.on('end', () => {
  server.close();
});
```

tls.connect(path[, options][, callback])

Added in: v0.11.3

- `path` <string> Default value for `options.path`.
- `options` <Object> See tls.connect().
- `callback` <Function> See tls.connect().

Same as tls.connect() except that `path` can be provided as an argument instead of an option.

A path option, if specified, will take precedence over the path argument.

tls.connect(port[, host][, options][, callback])

Added in: v0.11.3

- `port` <number> Default value for `options.port`.
- `host` <string> Optional default value for `options.host`.
- `options` <Object> See tls.connect().
- `callback` <Function> See tls.connect().

Same as tls.connect() except that `port` and `host` can be provided as arguments instead of options.

A port or host option, if specified, will take precedence over any port or host argument.

tls.createSecureContext(options)

▸ History

- `options` <Object>
  - `pfx` <string> | <string[]> | <Buffer> | <Buffer[]> | <Object[]> Optional PFX or PKCS12 encoded private key and certificate chain. `pfx` is an alternative to providing `key` and `cert` individually. PFX is usually encrypted, if it is, `passphrase` will be used to decrypt it. Multiple PFX can be provided either as an array of unencrypted PFX buffers, or an array of objects in the form `{buf: <string|buffer>[, passphrase: <string>]}`. The object form can only occur in an array. `object.passphrase` is optional. Encrypted PFX will be decrypted with `object.passphrase` if provided, or `options.passphrase` if it is not.
  - `key` <string> | <string[]> | <Buffer> | <Buffer[]> | <Object[]> Optional private keys in PEM format. PEM allows the option of private keys being encrypted. Encrypted keys will be decrypted with `options.passphrase`. Multiple keys using different algorithms can be provided either as an array of unencrypted key strings or buffers, or an array of objects in the form `{pem: <string|buffer>[, passphrase: <string>]}`. The object form can only occur in an array. `object.passphrase` is optional. Encrypted keys will be decrypted with `object.passphrase` if provided, or `options.passphrase` if it is not.
  - `passphrase` <string> Optional shared passphrase used for a single private key and/or a PFX.
  - `cert` <string> | <string[]> | <Buffer> | <Buffer[]> Optional cert chains in PEM format. One cert chain should be provided per private key. Each cert chain should consist of the PEM formatted certificate for a provided private `key`, followed by the PEM formatted intermediate certificates (if any), in order, and not including the root CA (the root CA must be pre-known to the peer, see `ca`). When providing multiple cert chains, they do not have to be in the same order as their private keys in `key`. If the intermediate certificates are not provided, the peer will not be able to validate the certificate, and the handshake will fail.
  - `ca` <string> | <string[]> | <Buffer> | <Buffer[]> Optionally override the trusted CA certificates. Default is to trust the well-known CAs curated by Mozilla. Mozilla's CAs are completely replaced when CAs are explicitly specified using this option. The value can be a string or Buffer, or an Array of strings and/or Buffers. Any string or Buffer can contain multiple PEM CAs concatenated together. The peer's certificate must be chainable to a CA trusted by the server for the connection to be authenticated. When using certificates that are not chainable to a well-known CA, the certificate's CA must be explicitly specified as a trusted or the connection will fail to authenticate. If the peer uses a certificate that doesn't match or chain to one of the default CAs, use the `ca` option to provide a CA certificate that the peer's certificate can match or chain to. For self-signed certificates, the certificate is its own CA, and must be provided.
  - `ciphers` <string> Optional cipher suite specification, replacing the default. For more information, see modifying the default cipher suite.
  - `honorCipherOrder` <boolean> Attempt to use the server's cipher suite preferences instead of the client's. When `true`, causes `SSL_OP_CIPHER_SERVER_PREFERENCE` to be set in `secureOptions`, see OpenSSL Options for more information.
  - `ecdhCurve` <string> A string describing a named curve or a colon separated list of curve NIDs or names, for example `P-521:P-384:P-256`, to use for ECDH key agreement, or `false` to disable ECDH. Set to `auto` to select the curve automatically. Defaults to tls.DEFAULT_ECDH_CURVE. Use crypto.getCurves() to obtain a list of available curve names. On recent releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve.
  - `clientCertEngine` <string> Optional name of an OpenSSL engine which can provide the client certificate.
  - `crl` <string> | <string[]> | <Buffer> | <Buffer[]> Optional PEM formatted CRLs (Certificate Revocation Lists).
  - `dhparam` <string> | <Buffer> Diffie Hellman parameters, required for Perfect Forward Secrecy. Use `openssl dhparam` to create the parameters. The key length must be greater than or equal to 1024 bits, otherwise an error will be thrown. It is strongly recommended to use 2048 bits or larger for stronger security. If omitted or invalid, the parameters are silently discarded and DHE

ciphers will not be available.
  - `secureOptions` <number> Optionally affect the OpenSSL protocol behavior, which is not usually necessary. This should be used carefully if at all! Value is a numeric bitmask of the `SSL_OP_*` options from OpenSSL Options.
  - `secureProtocol` <string> Optional SSL method to use, default is `"SSLv23_method"`. The possible values are listed as SSL_METHODS, use the function names as strings. For example, `"SSLv3_method"` to force SSL version 3.
  - `sessionIdContext` <string> Optional opaque identifier used by servers to ensure session state is not shared between applications. Unused by clients.

tls.createServer() sets the default value of the `honorCipherOrder` option to `true`, other APIs that create secure contexts leave it unset.

tls.createServer() uses a 128 bit truncated SHA1 hash value generated from `process.argv` as the default value of the `sessionIdContext` option, other APIs that create secure contexts have no default value.

The `tls.createSecureContext()` method creates a credentials object.

A key is *required* for ciphers that make use of certificates. Either `key` or `pfx` can be used to provide it.

If the 'ca' option is not given, then Node.js will use the default publicly trusted list of CAs as given in https://hg.mozilla.org/mozilla-central/raw-file/tip/security/nss/lib/ckfw/builtins/certdata.txt.

tls.createServer([options][, secureConnectionListener])

▸ History

- `options` <Object>
  - `clientCertEngine` <string> Optional name of an OpenSSL engine which can provide the client certificate.
  - `handshakeTimeout` <number> Abort the connection if the SSL/TLS handshake does not finish in the specified number of milliseconds. Defaults to `120000` (120 seconds). A `'tlsClientError'` is emitted on the `tls.Server` object whenever a handshake times out.
  - `requestCert` <boolean> If `true` the server will request a certificate from clients that connect and attempt to verify that certificate. Defaults to `false`.
  - `rejectUnauthorized` <boolean> If not `false` the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if `requestCert` is `true`. Defaults to `true`.
  - `NPNProtocols` <string[]> | <Buffer[]> | <Uint8Array[]> | <Buffer> | <Uint8Array> An array of strings, `Buffer`s or `Uint8Array`s, or a single `Buffer` or `Uint8Array` containing supported NPN protocols. `Buffer`s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g `['hello', 'world']`. (Protocols should be ordered by their priority.)
  - `ALPNProtocols`: <string[]> | <Buffer[]> | <Uint8Array[]> | <Buffer> | <Uint8Array> An array of strings, `Buffer`s or `Uint8Array`s, or a single `Buffer` or `Uint8Array` containing the supported ALPN protocols. `Buffer`s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g `['hello', 'world']`. (Protocols should be ordered by their priority.) When the server receives both NPN and ALPN extensions from the client, ALPN takes precedence over NPN and the server does not send an NPN extension to the client.
  - `SNICallback(servername, cb)` <Function> A function that will be called if the client supports SNI TLS extension. Two arguments will be passed when called: `servername` and `cb`. `SNICallback` should invoke `cb(null, ctx)`, where `ctx` is a SecureContext instance. (`tls.createSecureContext(...)` can be used to get a proper SecureContext.) If `SNICallback` wasn't provided the default callback with high-level API will be used (see below).
  - `sessionTimeout` <number> An integer specifying the number of seconds after which the TLS session identifiers and TLS session tickets created by the server will time out. See SSL_CTX_set_timeout for more details.
  - `ticketKeys`: A 48-byte `Buffer` instance consisting of a 16-byte prefix, a 16-byte HMAC key, and a 16-byte AES key. This can be used to accept TLS session tickets on multiple instances of the TLS server.
  - ...: Any tls.createSecureContext() options can be provided. For servers, the identity options (`pfx` or `key`/`cert`) are usually required.
- `secureConnectionListener` <Function>

Creates a new tls.Server. The `secureConnectionListener`, if provided, is automatically set as a listener for the 'secureConnection' event.

The `ticketKeys` options is automatically shared between `cluster` module workers.

The following illustrates a simple echo server:

```
const tls = require('tls');
const fs = require('fs');

const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),

  // This is necessary only if using the client certificate authentication.
  requestCert: true,

  // This is necessary only if the client uses the self-signed certificate.
  ca: [ fs.readFileSync('client-cert.pem') ]
};

const server = tls.createServer(options, (socket) => {
  console.log('server connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  socket.write('welcome!\n');
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, () => {
  console.log('server bound');
});
```

Or

```
const tls = require('tls');
const fs = require('fs');

const options = {
  pfx: fs.readFileSync('server.pfx'),

  // This is necessary only if using the client certificate authentication.
  requestCert: true,

};

const server = tls.createServer(options, (socket) => {
  console.log('server connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  socket.write('welcome!\n');
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, () => {
  console.log('server bound');
});
```

This server can be tested by connecting to it using `openssl s_client`:

```
openssl s_client -connect 127.0.0.1:8000
```

tls.getCiphers()

Added in: v0.10.2

Returns an array with the names of the supported SSL ciphers.

For example:

```
console.log(tls.getCiphers()); // ['AES128-SHA', 'AES256-SHA', ...]
```

tls.DEFAULT_ECDH_CURVE

Added in: v0.11.13

The default curve name to use for ECDH key agreement in a tls server. The default value is 'prime256v1' (NIST P-256). Consult RFC 4492 and FIPS.186-4 for more details.

Deprecated APIs

Class: CryptoStream

Added in: v0.3.4 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use tls.TLSSocket instead.

The tls.CryptoStream class represents a stream of encrypted data. This class has been deprecated and should no longer be used.

cryptoStream.bytesWritten

Added in: v0.3.4 Deprecated since: v0.11.3

The cryptoStream.bytesWritten property returns the total number of bytes written to the underlying socket *including* the bytes required for the implementation of the TLS protocol.

Class: SecurePair

Added in: v0.3.2 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use tls.TLSSocket instead.

Returned by tls.createSecurePair() .

Event: 'secure'

Added in: v0.3.2 Deprecated since: v0.11.3

The 'secure' event is emitted by the SecurePair object once a secure connection has been established.

As with checking for the server secureConnection event, pair.cleartext.authorized should be inspected to confirm whether the certificate used is properly authorized.

tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])

▶ History

Stability: 0 - Deprecated: Use tls.TLSSocket instead.

- context <Object> A secure context object as returned by tls.createSecureContext()
- isServer <boolean> true to specify that this TLS connection should be opened as a server.

- `requestCert` <boolean> `true` to specify whether a server should request a certificate from a connecting client. Only applies when `isServer` is `true`.
- `rejectUnauthorized` <boolean> If not `false` a server automatically reject clients with invalid certificates. Only applies when `isServer` is `true`.
- `options`
  - `secureContext`: An optional TLS context object from tls.createSecureContext()
  - `isServer`: If `true` the TLS socket will be instantiated in server-mode. Defaults to `false`.
  - `server` <net.Server> An optional net.Server instance
  - `requestCert`: Optional, see tls.createServer()
  - `rejectUnauthorized`: Optional, see tls.createServer()
  - `NPNProtocols`: Optional, see tls.createServer()
  - `ALPNProtocols`: Optional, see tls.createServer()
  - `SNICallback`: Optional, see tls.createServer()
  - `session` <Buffer> An optional `Buffer` instance containing a TLS session.
  - `requestOCSP` <boolean> If `true`, specifies that the OCSP status request extension will be added to the client hello and an `'OCSPResponse'` event will be emitted on the socket before establishing a secure communication

Creates a new secure pair object with two streams, one of which reads and writes the encrypted data and the other of which reads and writes the cleartext data. Generally, the encrypted stream is piped to/from an incoming encrypted data stream and the cleartext one is used as a replacement for the initial encrypted stream.

`tls.createSecurePair()` returns a `tls.SecurePair` object with `cleartext` and `encrypted` stream properties.

Using `cleartext` has the same API as tls.TLSSocket.

The `tls.createSecurePair()` method is now deprecated in favor of `tls.TLSSocket()`. For example, the code:

```
pair = tls.createSecurePair(/* ... */);
pair.encrypted.pipe(socket);
socket.pipe(pair.encrypted);
```

can be replaced by:

```
secure_socket = tls.TLSSocket(socket, options);
```

where `secure_socket` has the same API as `pair.cleartext`.

Tracing

Trace Event provides a mechanism to centralize tracing information generated by V8, Node core, and userspace code.

Tracing can be enabled by passing the `--trace-events-enabled` flag when starting a Node.js application.

The set of categories for which traces are recorded can be specified using the `--trace-event-categories` flag followed by a list of comma separated category names. By default the `node`, `node.async_hooks`, and `v8` categories are enabled.

```
node --trace-events-enabled --trace-event-categories v8,node,node.async_hooks server.js
```

Running Node.js with tracing enabled will produce log files that can be opened in the chrome://tracing tab of Chrome.

The logging file is by default called `node_trace.${rotation}.log`, where `${rotation}` is an incrementing log-rotation id. The filepath pattern can be specified with `--trace-event-file-pattern` that accepts a template string that supports `${rotation}` and `${pid}`. For example:

```
node --trace-events-enabled --trace-event-file-pattern '${pid}-${rotation}.log' server.js
```

Starting with Node 10.0.0, the tracing system uses the same time source as the one used by `process.hrtime()` however the trace-event timestamps are expressed in microseconds, unlike `process.hrtime()` which returns nanoseconds.

TTY

Stability: 2 - Stable

The `tty` module provides the `tty.ReadStream` and `tty.WriteStream` classes. In most cases, it will not be necessary or possible to use this module directly. However, it can be accessed using:

```
const tty = require('tty');
```

When Node.js detects that it is being run with a text terminal ("TTY") attached, `process.stdin` will, by default, be initialized as an instance of `tty.ReadStream` and both `process.stdout` and `process.stderr` will, by default be instances of `tty.WriteStream`. The preferred method of determining whether Node.js is being run within a TTY context is to check that the value of the `process.stdout.isTTY` property is `true`:

```
$ node -p -e "Boolean(process.stdout.isTTY)"
true
$ node -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

In most cases, there should be little to no reason for an application to manually create instances of the `tty.ReadStream` and `tty.WriteStream` classes.

Class: tty.ReadStream

Added in: v0.5.8

The `tty.ReadStream` class is a subclass of `net.Socket` that represents the readable side of a TTY. In normal circumstances `process.stdin` will be the only `tty.ReadStream` instance in a Node.js process and there should be no reason to create additional instances.

readStream.isRaw

Added in: v0.7.7

A `boolean` that is `true` if the TTY is currently configured to operate as a raw device. Defaults to `false`.

readStream.isTTY

Added in: v0.5.8

A `boolean` that is always `true` for `tty.ReadStream` instances.

readStream.setRawMode(mode)

Added in: v0.7.7

Allows configuration of `tty.ReadStream` so that it operates as a raw device.

When in raw mode, input is always available character-by-character, not including modifiers. Additionally, all special processing of characters by the terminal is disabled, including echoing input characters. Note that `CTRL+C` will no longer cause a `SIGINT` when in this mode.

- `mode` <boolean> If `true`, configures the `tty.ReadStream` to operate as a raw device. If `false`, configures the `tty.ReadStream` to operate in its default mode. The `readStream.isRaw` property will be set to the resulting mode.

Class: tty.WriteStream

Added in: v0.5.8

The `tty.WriteStream` class is a subclass of `net.Socket` that represents the writable side of a TTY. In normal circumstances, `process.stdout` and `process.stderr` will be the only `tty.WriteStream` instances created for a Node.js process and there should be no reason to create additional instances.

Event: 'resize'

Added in: v0.7.7

The `'resize'` event is emitted whenever either of the `writeStream.columns` or `writeStream.rows` properties have changed. No arguments are passed to the listener callback when called.

```
process.stdout.on('resize', () => {
  console.log('screen size has changed!');
  console.log(`${process.stdout.columns}x${process.stdout.rows}`);
});
```

writeStream.columns

Added in: v0.7.7

A `number` specifying the number of columns the TTY currently has. This property is updated whenever the `'resize'` event is emitted.

writeStream.isTTY

Added in: v0.5.8

A `boolean` that is always `true`.

writeStream.rows

Added in: v0.7.7

A `number` specifying the number of rows the TTY currently has. This property is updated whenever the `'resize'` event is emitted.

writeStream.getColorDepth([env])

Added in: v9.9.0

- `env` <object> A object containing the environment variables to check. Defaults to `process.env`.
- Returns: <number>

Returns:

- 1 for 2,
- 4 for 16,
- 8 for 256,
- 24 for 16,777,216 colors supported.

Use this to determine what colors the terminal supports. Due to the nature of colors in terminals it is possible to either have false positives or false negatives. It depends on process information and the environment variables that may lie about what terminal is used. To enforce a specific behavior without relying on `process.env` it is possible to pass in an object with different settings.

Use the `NODE_DISABLE_COLORS` environment variable to enforce this function to always return 1.

tty.isatty(fd)

Added in: v0.5.8

- `fd` <number> A numeric file descriptor

The `tty.isatty()` method returns `true` if the given `fd` is associated with a TTY and `false` if it is not, including whenever `fd` is not a non-negative integer.

UDP / Datagram Sockets

Stability: 2 - Stable

The `dgram` module provides an implementation of UDP Datagram sockets.

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

Class: dgram.Socket

Added in: v0.1.99

The `dgram.Socket` object is an EventEmitter that encapsulates the datagram functionality.

New instances of `dgram.Socket` are created using `dgram.createSocket()`. The `new` keyword is not to be used to create `dgram.Socket` instances.

Event: 'close'

Added in: v0.1.99

The `'close'` event is emitted after a socket is closed with `close()`. Once triggered, no new `'message'` events will be emitted on this socket.

Event: 'error'

Added in: v0.1.99

- `exception` <Error>

The `'error'` event is emitted whenever any error occurs. The event handler function is passed a single Error object.

Event: 'listening'

Added in: v0.1.99

The `'listening'` event is emitted whenever a socket begins listening for datagram messages. This occurs as soon as UDP sockets are created.

Event: 'message'

Added in: v0.1.99

The `'message'` event is emitted when a new datagram is available on a socket. The event handler function is passed two arguments: `msg` and

`rinfo`.

- `msg` <Buffer> The message.
- `rinfo` <Object> Remote address information.
  - `address` <string> The sender address.
  - `family` <string> The address family (`'IPv4'` or `'IPv6'`).
  - `port` <number> The sender port.
  - `size` <number> The message size.

socket.addMembership(multicastAddress[, multicastInterface])

Added in: v0.6.9

- `multicastAddress` <string>
- `multicastInterface` <string>

Tells the kernel to join a multicast group at the given `multicastAddress` and `multicastInterface` using the `IP_ADD_MEMBERSHIP` socket option. If the `multicastInterface` argument is not specified, the operating system will choose one interface and will add membership to it. To add membership to every available interface, call `addMembership` multiple times, once per interface.

socket.address()

Added in: v0.1.99

Returns an object containing the address information for a socket. For UDP sockets, this object will contain `address`, `family` and `port` properties.

socket.bind([port][, address][, callback])

Added in: v0.1.99

- `port` <number> Integer.
- `address` <string>
- `callback` <Function> with no parameters. Called when binding is complete.

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address`. If `port` is not specified or is `0`, the operating system will attempt to bind to a random port. If `address` is not specified, the operating system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

Note that specifying both a `'listening'` event listener and passing a `callback` to the `socket.bind()` method is not harmful but not very useful.

A bound datagram socket keeps the Node.js process running to receive datagram messages.

If binding fails, an `'error'` event is generated. In rare case (e.g. attempting to bind with a closed socket), an Error may be thrown.

Example of a UDP server listening on port 41234:

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

socket.bind(options[, callback])

Added in: v0.11.14

- `options` <Object> Required. Supports the following properties:
  - `port` <integer>
  - `address` <string>
  - `exclusive` <boolean>
- `callback` <Function>

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address` that are passed as properties of an `options` object passed as the first argument. If `port` is not specified or is `0`, the operating system will attempt to bind to a random port. If `address` is not specified, the operating system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

Note that specifying both a `'listening'` event listener and passing a `callback` to the `socket.bind()` method is not harmful but not very useful.

The `options` object may contain an additional `exclusive` property that is use when using `dgram.Socket` objects with the cluster module. When `exclusive` is set to `false` (the default), cluster workers will use the same underlying socket handle allowing connection handling duties to be shared. When `exclusive` is `true`, however, the handle is not shared and attempted port sharing results in an error.

A bound datagram socket keeps the Node.js process running to receive datagram messages.

If binding fails, an `'error'` event is generated. In rare case (e.g. attempting to bind with a closed socket), an Error may be thrown.

An example socket listening on an exclusive port is shown below.

```
socket.bind({
  address: 'localhost',
  port: 8000,
  exclusive: true
});
```

socket.close([callback])

Added in: v0.1.99

Close the underlying socket and stop listening for data on it. If a callback is provided, it is added as a listener for the 'close' event.

socket.dropMembership(multicastAddress[, multicastInterface])

Added in: v0.6.9

- multicastAddress <string>
- multicastInterface <string>

Instructs the kernel to leave a multicast group at multicastAddress using the IP_DROP_MEMBERSHIP socket option. This method is automatically called by the kernel when the socket is closed or the process terminates, so most apps will never have reason to call this.

If multicastInterface is not specified, the operating system will attempt to drop membership on all valid interfaces.

socket.getRecvBufferSize()

Added in: v8.7.0

- Returns: <number> the SO_RCVBUF socket receive buffer size in bytes.

socket.getSendBufferSize()

Added in: v8.7.0

- Returns: <number> the SO_SNDBUF socket send buffer size in bytes.

socket.ref()

Added in: v0.9.1

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The socket.unref() method can be used to exclude the socket from the reference counting that keeps the Node.js process active. The socket.ref() method adds the socket back to the reference counting and restores the default behavior.

Calling socket.ref() multiples times will have no additional effect.

The socket.ref() method returns a reference to the socket so calls can be chained.

socket.send(msg, [offset, length,] port [, address] [, callback])

▸ History

- msg <Buffer> | <Uint8Array> | <string> | <Array> Message to be sent.
- offset <number> Integer. Offset in the buffer where the message starts.
- length <number> Integer. Number of bytes in the message.
- port <number> Integer. Destination port.
- address <string> Destination hostname or IP address.
- callback <Function> Called when the message has been sent.

Broadcasts a datagram on the socket. The destination port and address must be specified.

The msg argument contains the message to be sent. Depending on its type, different behavior can apply. If msg is a Buffer or Uint8Array, the offset and length specify the offset within the Buffer where the message begins and the number of bytes in the message, respectively. If msg is a String, then it is automatically converted to a Buffer with 'utf8' encoding. With messages that contain multi-byte characters, offset and length will be calculated with respect to byte length and not the character position. If msg is an array, offset and length must not be specified.

The address argument is a string. If the value of address is a host name, DNS will be used to resolve the address of the host. If address is not provided or otherwise falsy, '127.0.0.1' (for udp4 sockets) or '::1' (for udp6 sockets) will be used by default.

If the socket has not been previously bound with a call to bind, the socket is assigned a random port number and is bound to the "all interfaces" address ('0.0.0.0' for udp4 sockets, '::0' for udp6 sockets.)

An optional `callback` function may be specified to as a way of reporting DNS errors or for determining when it is safe to reuse the `buf` object. Note that DNS lookups delay the time to send for at least one tick of the Node.js event loop.

The only way to know for sure that the datagram has been sent is by using a `callback`. If an error occurs and a `callback` is given, the error will be passed as the first argument to the `callback`. If a `callback` is not given, the error is emitted as an `'error'` event on the `socket` object.

Offset and length are optional but both *must* be set if either are used. They are supported only when the first argument is a `Buffer` or `Uint8Array`.

Example of sending a UDP packet to a port on `localhost`;

```
const dgram = require('dgram');
const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
  client.close();
});
```

Example of sending a UDP packet composed of multiple buffers to a port on `127.0.0.1`;

```
const dgram = require('dgram');
const buf1 = Buffer.from('Some ');
const buf2 = Buffer.from('bytes');
const client = dgram.createSocket('udp4');
client.send([buf1, buf2], 41234, (err) => {
  client.close();
});
```

Sending multiple buffers might be faster or slower depending on the application and operating system. It is important to run benchmarks to determine the optimal strategy on a case-by-case basis. Generally speaking, however, sending multiple buffers is faster.

A Note about UDP datagram size

The maximum size of an `IPv4/v6` datagram depends on the `MTU` (*Maximum Transmission Unit*) and on the `Payload Length` field size.

- The `Payload Length` field is `16 bits` wide, which means that a normal payload exceed 64K octets *including* the internet header and data (65,507 bytes = 65,535 − 8 bytes UDP header − 20 bytes IP header); this is generally true for loopback interfaces, but such long datagram messages are impractical for most hosts and networks.

- The `MTU` is the largest size a given link layer technology can support for datagram messages. For any link, `IPv4` mandates a minimum `MTU` of `68` octets, while the recommended `MTU` for IPv4 is `576` (typically recommended as the `MTU` for dial-up type applications), whether they arrive whole or in fragments.

  For `IPv6`, the minimum `MTU` is `1280` octets, however, the mandatory minimum fragment reassembly buffer size is `1500` octets. The value of `68` octets is very small, since most current link layer technologies, like Ethernet, have a minimum `MTU` of `1500`.

It is impossible to know in advance the MTU of each link through which a packet might travel. Sending a datagram greater than the receiver `MTU` will not work because the packet will get silently dropped without informing the source that the data did not reach its intended recipient.

socket.setBroadcast(flag)

Added in: v0.6.9

- `flag` <boolean>

Sets or clears the `SO_BROADCAST` socket option. When set to `true`, UDP packets may be sent to a local interface's broadcast address.

socket.setMulticastInterface(multicastInterface)

Added in: v8.6.0

- `multicastInterface` <string>

Note: All references to scope in this section are referring to IPv6 Zone Indices, which are defined by RFC 4007. In string form, an IP with a scope index is written as `'IP%scope'` where scope is an interface name or interface number.

Sets the default outgoing multicast interface of the socket to a chosen interface or back to system interface selection. The `multicastInterface` must be a valid string representation of an IP from the socket's family.

For IPv4 sockets, this should be the IP configured for the desired physical interface. All packets sent to multicast on the socket will be sent on the interface determined by the most recent successful use of this call.

For IPv6 sockets, `multicastInterface` should include a scope to indicate the interface as in the examples that follow. In IPv6, individual `send` calls can also use explicit scope in addresses, so only packets sent to a multicast address without specifying an explicit scope are affected by the most recent successful use of this call.

Examples: IPv6 Outgoing Multicast Interface

On most systems, where scope format uses the interface name:

```
const socket = dgram.createSocket('udp6');

socket.bind(1234, () => {
  socket.setMulticastInterface('::%eth1');
});
```

On Windows, where scope format uses an interface number:

```
const socket = dgram.createSocket('udp6');

socket.bind(1234, () => {
  socket.setMulticastInterface('::%2');
});
```

Example: IPv4 Outgoing Multicast Interface

All systems use an IP of the host on the desired physical interface:

```
const socket = dgram.createSocket('udp4');

socket.bind(1234, () => {
  socket.setMulticastInterface('10.0.0.2');
});
```

Call Results

A call on a socket that is not ready to send or no longer open may throw a *Not running* Error.

If `multicastInterface` can not be parsed into an IP then an *EINVAL* System Error is thrown.

On IPv4, if `multicastInterface` is a valid address but does not match any interface, or if the address does not match the family then a System Error such as `EADDRNOTAVAIL` or `EPROTONOSUP` is thrown.

On IPv6, most errors with specifying or omitting scope will result in the socket continuing to use (or returning to) the system's default interface selection.

A socket's address family's ANY address (IPv4 `'0.0.0.0'` or IPv6 `'::'`) can be used to return control of the sockets default outgoing interface to the system for future multicast packets.

socket.setMulticastLoopback(flag)

Added in: v0.3.8

- `flag` <boolean>

Sets or clears the `IP_MULTICAST_LOOP` socket option. When set to `true`, multicast packets will also be received on the local interface.

socket.setMulticastTTL(ttl)

Added in: v0.3.8

- `ttl` <number> Integer.

Sets the `IP_MULTICAST_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel through, specifically for multicast traffic. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded.

The argument passed to `socket.setMulticastTTL()` is a number of hops between 0 and 255. The default on most systems is `1` but can vary.

socket.setRecvBufferSize(size)

Added in: v8.7.0

- `size` <number> Integer

Sets the `SO_RCVBUF` socket option. Sets the maximum socket receive buffer in bytes.

socket.setSendBufferSize(size)

Added in: v8.7.0

- `size` <number> Integer

Sets the `SO_SNDBUF` socket option. Sets the maximum socket send buffer in bytes.

socket.setTTL(ttl)

Added in: v0.1.101

- `ttl` <number> Integer.

Sets the `IP_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel through. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded. Changing TTL values is typically done for network probes or when multicasting.

The argument to `socket.setTTL()` is a number of hops between 1 and 255. The default on most systems is 64 but can vary.

socket.unref()

Added in: v0.9.1

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The `socket.unref()` method can be used to exclude the socket from the reference counting that keeps the Node.js process active, allowing the process to exit even if the socket is still listening.

Calling `socket.unref()` multiple times will have no addition effect.

The `socket.unref()` method returns a reference to the socket so calls can be chained.

Change to asynchronous `socket.bind()` behavior

As of Node.js v0.10, dgram.Socket#bind() changed to an asynchronous execution model. Legacy code that assumes synchronous behavior, as in the following example:

```
const s = dgram.createSocket('udp4');
s.bind(1234);
s.addMembership('224.0.0.114');
```

Must be changed to pass a callback function to the dgram.Socket#bind() function:

```
const s = dgram.createSocket('udp4');
s.bind(1234, () => {
  s.addMembership('224.0.0.114');
});
```

dgram module functions

dgram.createSocket(options[, callback])

▸ History

- options <Object> Available options are:
  - type <string> The family of socket. Must be either 'udp4' or 'udp6'. Required.
  - reuseAddr <boolean> When true socket.bind() will reuse the address, even if another process has already bound a socket on it. Defaults to false.
  - recvBufferSize <number> - Sets the SO_RCVBUF socket value.
  - sendBufferSize <number> - Sets the SO_SNDBUF socket value.
  - lookup <Function> Custom lookup function. Defaults to dns.lookup().
- callback <Function> Attached as a listener for 'message' events. Optional.
- Returns: <dgram.Socket>

Creates a dgram.Socket object. Once the socket is created, calling socket.bind() will instruct the socket to begin listening for datagram messages. When address and port are not passed to socket.bind() the method will bind the socket to the "all interfaces" address on a random port (it does the right thing for both udp4 and udp6 sockets). The bound address and port can be retrieved using socket.address().address and socket.address().port.

dgram.createSocket(type[, callback])

Added in: v0.1.99

- type <string> - Either 'udp4' or 'udp6'.
- callback <Function> - Attached as a listener to 'message' events.
- Returns: <dgram.Socket>

Creates a dgram.Socket object of the specified type. The type argument can be either udp4 or udp6. An optional callback function can be passed which is added as a listener for 'message' events.

Once the socket is created, calling socket.bind() will instruct the socket to begin listening for datagram messages. When address and port are not passed to socket.bind() the method will bind the socket to the "all interfaces" address on a random port (it does the right thing for both udp4 and udp6 sockets). The bound address and port can be retrieved using socket.address().address and socket.address().port.

URL

Stability: 2 - Stable

The url module provides utilities for URL resolution and parsing. It can be accessed using:

```
const url = require('url');
```

URL Strings and URL Objects

A URL string is a structured string containing multiple meaningful components. When parsed, a URL object is returned containing properties for each of these components.

The `url` module provides two APIs for working with URLs: a legacy API that is Node.js specific, and a newer API that implements the same WHATWG URL Standard used by web browsers.

While the Legacy API has not been deprecated, it is maintained solely for backwards compatibility with existing applications. New application code should use the WHATWG API.

A comparison between the WHATWG and Legacy APIs is provided below. Above the URL `'http://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash'`, properties of an object returned by the legacy `url.parse()` are shown. Below it are properties of a WHATWG `URL` object.

WHATWG URL's `origin` property includes `protocol` and `host`, but not `username` or `password`.

```
┌────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                              href                                              │
├──────────┬──┬─────────────────────┬────────────────────────┬───────────────────────────┬───────┤
│ protocol │  │        auth         │          host          │           path            │ hash  │
│          │  │                     ├─────────────────┬──────┼──────────┬────────────────┤       │
│          │  │                     │    hostname     │ port │ pathname │     search     │       │
│          │  │                     │                 │      │          ├─┬──────────────┤       │
│          │  │                     │                 │      │          │ │    query     │       │
"  https:   //    user   :   pass   @ sub.host.com : 8080   /p/a/t/h  ?  query=string   #hash "
│          │  │          │          │    hostname     │ port │          │                │       │
│          │  │          │          ├─────────────────┴──────┤          │                │       │
│ protocol │  │ username │ password │          host          │          │                │       │
├──────────┴──┼──────────┴──────────┼────────────────────────┤          │                │       │
│   origin    │                     │         origin         │ pathname │     search     │ hash  │
├─────────────┴─────────────────────┴────────────────────────┴──────────┴────────────────┴───────┤
│                                              href                                              │
└────────────────────────────────────────────────────────────────────────────────────────────────┘
(all spaces in the "" line should be ignored — they are purely for formatting)
```

Parsing the URL string using the WHATWG API:

```
const { URL } = require('url');
const myURL =
  new URL('https://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash');
```

Note: In Web Browsers, the WHATWG `URL` class is a global that is always available. In Node.js, however, the `URL` class must be accessed via `require('url').URL`.

Parsing the URL string using the Legacy API:

```
const url = require('url');
const myURL =
  url.parse('https://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash');
```

The WHATWG URL API

Added in: v7.0.0

Class: URL

Browser-compatible `URL` class, implemented by following the WHATWG URL Standard. Examples of parsed URLs may be found in the Standard itself.

In accordance with browser conventions, all properties of `URL` objects are implemented as getters and setters on the class prototype, rather than as data properties on the object itself. Thus, unlike legacy urlObject s, using the `delete` keyword on any properties of `URL` objects (e.g. `delete myURL.protocol`, `delete myURL.pathname`, etc) has no effect but will still return `true`.

Constructor: new URL(input[, base])

- `input` <string> The input URL to parse
- `base` <string> | <URL> The base URL to resolve against if the `input` is not absolute.

Creates a new `URL` object by parsing the `input` relative to the `base`. If `base` is passed as a string, it will be parsed equivalent to `new URL(base)`.

```
const { URL } = require('url');
const myURL = new URL('/foo', 'https://example.org/');
// https://example.org/foo
```

A `TypeError` will be thrown if the `input` or `base` are not valid URLs. Note that an effort will be made to coerce the given values into strings. For instance:

```
const { URL } = require('url');
const myURL = new URL({ toString: () => 'https://example.org/' });
// https://example.org/
```

Unicode characters appearing within the hostname of `input` will be automatically converted to ASCII using the Punycode algorithm.

```
const { URL } = require('url');
const myURL = new URL('https://□□□□');
// https://xn--6qqa088eba/
```

This feature is only available if the `node` executable was compiled with ICU enabled. If not, the domain names are passed through unchanged.

url.hash

- <string>

Gets and sets the fragment portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://example.org/foo#bar');
console.log(myURL.hash);
// Prints #bar

myURL.hash = 'baz';
console.log(myURL.href);
// Prints https://example.org/foo#baz
```

Invalid URL characters included in the value assigned to the `hash` property are percent-encoded . Note that the selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

url.host

- <string>

Gets and sets the host portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.host);
// Prints example.org:81

myURL.host = 'example.com:82';
console.log(myURL.href);
// Prints https://example.com:82/foo
```

Invalid host values assigned to the `host` property are ignored.

url.hostname

- <string>

Gets and sets the hostname portion of the URL. The key difference between `url.host` and `url.hostname` is that `url.hostname` does *not* include the port.

```
const { URL } = require('url');
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.hostname);
// Prints example.org

myURL.hostname = 'example.com:82';
console.log(myURL.href);
// Prints https://example.com:81/foo
```

Invalid hostname values assigned to the `hostname` property are ignored.

url.href

- <string>

Gets and sets the serialized URL.

```
const { URL } = require('url');
const myURL = new URL('https://example.org/foo');
console.log(myURL.href);
// Prints https://example.org/foo

myURL.href = 'https://example.com/bar';
console.log(myURL.href);
// Prints https://example.com/bar
```

Getting the value of the `href` property is equivalent to calling url.toString() .

Setting the value of this property to a new value is equivalent to creating a new `URL` object using new URL(value) . Each of the `URL` object's properties will be modified.

If the value assigned to the `href` property is not a valid URL, a `TypeError` will be thrown.

url.origin

- <string>

Gets the read-only serialization of the URL's origin.

```
const { URL } = require('url');
const myURL = new URL('https://example.org/foo/bar?baz');
console.log(myURL.origin);
// Prints https://example.org
```

```
const { URL } = require('url');
const idnURL = new URL('https://□□□□');
console.log(idnURL.origin);
// Prints https://xn--6qqa088eba

console.log(idnURL.hostname);
// Prints xn--6qqa088eba
```

url.password

- <string>

Gets and sets the password portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://abc:xyz@example.com');
console.log(myURL.password);
// Prints xyz

myURL.password = '123';
console.log(myURL.href);
// Prints https://abc:123@example.com
```

Invalid URL characters included in the value assigned to the `password` property are percent-encoded . Note that the selection of which characters to percent-encode may vary somewhat from what the url.parse() and url.format() methods would produce.

url.pathname

- <string>

Gets and sets the path portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://example.org/abc/xyz?123');
console.log(myURL.pathname);
// Prints /abc/xyz

myURL.pathname = '/abcdef';
console.log(myURL.href);
// Prints https://example.org/abcdef?123
```

Invalid URL characters included in the value assigned to the `pathname` property are percent-encoded . Note that the selection of which characters to percent-encode may vary somewhat from what the url.parse() and url.format() methods would produce.

url.port

- <string>

Gets and sets the port portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://example.org:8888');
console.log(myURL.port);
// Prints 8888

// Default ports are automatically transformed to the empty string
// (HTTPS protocol's default port is 443)
myURL.port = '443';
console.log(myURL.port);
// Prints the empty string
console.log(myURL.href);
// Prints https://example.org/

myURL.port = 1234;
console.log(myURL.port);
// Prints 1234
console.log(myURL.href);
// Prints https://example.org:1234/

// Completely invalid port strings are ignored
myURL.port = 'abcd';
console.log(myURL.port);
// Prints 1234

// Leading numbers are treated as a port number
myURL.port = '5678abcd';
console.log(myURL.port);
// Prints 5678

// Non-integers are truncated
myURL.port = 1234.5678;
console.log(myURL.port);
// Prints 1234

// Out-of-range numbers are ignored
myURL.port = 1e10;
console.log(myURL.port);
// Prints 1234
```

The port value may be set as either a number or as a String containing a number in the range `0` to `65535` (inclusive). Setting the value to the default port of the `URL` objects given `protocol` will result in the `port` value becoming the empty string (`''`).

If an invalid string is assigned to the `port` property, but it begins with a number, the leading number is assigned to `port`. Otherwise, or if the number lies outside the range denoted above, it is ignored.

url.protocol

- <string>

Gets and sets the protocol portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://example.org');
console.log(myURL.protocol);
// Prints https:

myURL.protocol = 'ftp';
console.log(myURL.href);
// Prints ftp://example.org/
```

Invalid URL protocol values assigned to the `protocol` property are ignored.

url.search

- <string>

Gets and sets the serialized query portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://example.org/abc?123');
console.log(myURL.search);
// Prints ?123

myURL.search = 'abc=xyz';
console.log(myURL.href);
// Prints https://example.org/abc?abc=xyz
```

Any invalid URL characters appearing in the value assigned the `search` property will be percent-encoded . Note that the selection of which characters to percent-encode may vary somewhat from what the url.parse() and url.format() methods would produce.

url.searchParams

- <URLSearchParams>

Gets the URLSearchParams object representing the query parameters of the URL. This property is read-only; to replace the entirety of query parameters of the URL, use the url.search setter. See URLSearchParams documentation for details.

url.username

- <string>

Gets and sets the username portion of the URL.

```
const { URL } = require('url');
const myURL = new URL('https://abc:xyz@example.com');
console.log(myURL.username);
// Prints abc

myURL.username = '123';
console.log(myURL.href);
// Prints https://123:xyz@example.com/
```

Any invalid URL characters appearing in the value assigned the `username` property will be percent-encoded . Note that the selection of which characters to percent-encode may vary somewhat from what the url.parse() and url.format() methods would produce.

url.toString()

- Returns: <string>

The `toString()` method on the `URL` object returns the serialized URL. The value returned is equivalent to that of url.href and url.toJSON() .

Because of the need for standard compliance, this method does not allow users to customize the serialization process of the URL. For more flexibility, require('url').format() method might be of interest.

url.toJSON()

- Returns: <string>

The toJSON() method on the URL object returns the serialized URL. The value returned is equivalent to that of url.href and url.toString() .

This method is automatically called when an URL object is serialized with JSON.stringify() .

```
const { URL } = require('url');
const myURLs = [
  new URL('https://www.example.com'),
  new URL('https://test.example.org')
];
console.log(JSON.stringify(myURLs));
// Prints ["https://www.example.com/","https://test.example.org/"]
```

Class: URLSearchParams

Added in: v7.5.0

The URLSearchParams API provides read and write access to the query of a URL . The URLSearchParams class can also be used standalone with one of the four following constructors.

The WHATWG URLSearchParams interface and the querystring module have similar purpose, but the purpose of the querystring module is more general, as it allows the customization of delimiter characters (& and =). On the other hand, this API is designed purely for URL query strings.

```
const { URL, URLSearchParams } = require('url');

const myURL = new URL('https://example.org/?abc=123');
console.log(myURL.searchParams.get('abc'));
// Prints 123

myURL.searchParams.append('abc', 'xyz');
console.log(myURL.href);
// Prints https://example.org/?abc=123&abc=xyz

myURL.searchParams.delete('abc');
myURL.searchParams.set('a', 'b');
console.log(myURL.href);
// Prints https://example.org/?a=b

const newSearchParams = new URLSearchParams(myURL.searchParams);
// The above is equivalent to
// const newSearchParams = new URLSearchParams(myURL.search);

newSearchParams.append('a', 'c');
console.log(myURL.href);
// Prints https://example.org/?a=b
console.log(newSearchParams.toString());
// Prints a=b&a=c

// newSearchParams.toString() is implicitly called
myURL.search = newSearchParams;
console.log(myURL.href);
// Prints https://example.org/?a=b&a=c
newSearchParams.delete('a');
console.log(myURL.href);
// Prints https://example.org/?a=b&a=c
```

Constructor: new URLSearchParams()

Instantiate a new empty `URLSearchParams` object.

Constructor: new URLSearchParams(string)

- `string` <string> A query string

Parse the `string` as a query string, and use it to instantiate a new `URLSearchParams` object. A leading `'?'`, if present, is ignored.

```
const { URLSearchParams } = require('url');
let params;

params = new URLSearchParams('user=abc&query=xyz');
console.log(params.get('user'));
// Prints 'abc'
console.log(params.toString());
// Prints 'user=abc&query=xyz'

params = new URLSearchParams('?user=abc&query=xyz');
console.log(params.toString());
// Prints 'user=abc&query=xyz'
```

Constructor: new URLSearchParams(obj)

Added in: v7.10.0

- `obj` <Object> An object representing a collection of key-value pairs

Instantiate a new `URLSearchParams` object with a query hash map. The key and value of each property of `obj` are always coerced to strings.

Unlike `querystring` module, duplicate keys in the form of array values are not allowed. Arrays are stringified using `array.toString()`, which simply joins all array elements with commas.

```
const { URLSearchParams } = require('url');
const params = new URLSearchParams({
  user: 'abc',
  query: ['first', 'second']
});
console.log(params.getAll('query'));
// Prints [ 'first,second' ]
console.log(params.toString());
// Prints 'user=abc&query=first%2Csecond'
```

Constructor: new URLSearchParams(iterable)

Added in: v7.10.0

- `iterable` <Iterable> An iterable object whose elements are key-value pairs

Instantiate a new `URLSearchParams` object with an iterable map in a way that is similar to `Map`'s constructor. `iterable` can be an Array or any iterable object. That means `iterable` can be another `URLSearchParams`, in which case the constructor will simply create a clone of the provided `URLSearchParams`. Elements of `iterable` are key-value pairs, and can themselves be any iterable object.

Duplicate keys are allowed.

```
const { URLSearchParams } = require('url');
let params;

// Using an array
params = new URLSearchParams([
  ['user', 'abc'],
  ['query', 'first'],
  ['query', 'second']
]);
console.log(params.toString());
// Prints 'user=abc&query=first&query=second'

// Using a Map object
const map = new Map();
map.set('user', 'abc');
map.set('query', 'xyz');
params = new URLSearchParams(map);
console.log(params.toString());
// Prints 'user=abc&query=xyz'

// Using a generator function
function* getQueryPairs() {
  yield ['user', 'abc'];
  yield ['query', 'first'];
  yield ['query', 'second'];
}
params = new URLSearchParams(getQueryPairs());
console.log(params.toString());
// Prints 'user=abc&query=first&query=second'

// Each key-value pair must have exactly two elements
new URLSearchParams([
  ['user', 'abc', 'error']
]);
// Throws TypeError [ERR_INVALID_TUPLE]:
//        Each query pair must be an iterable [name, value] tuple
```

urlSearchParams.append(name, value)

- `name` <string>
- `value` <string>

Append a new name-value pair to the query string.

urlSearchParams.delete(name)

- `name` <string>

Remove all name-value pairs whose name is `name`.

urlSearchParams.entries()

- Returns: <Iterator>

Returns an ES6 Iterator over each of the name-value pairs in the query. Each item of the iterator is a JavaScript Array. The first item of the Array is the `name`, the second item of the Array is the `value`.

Alias for urlSearchParams[@@iterator]().

urlSearchParams.forEach(fn[, thisArg])

- `fn` <Function> Function invoked for each name-value pair in the query.
- `thisArg` <Object> Object to be used as `this` value for when `fn` is called

Iterates over each name-value pair in the query and invokes the given function.

```
const { URL } = require('url');
const myURL = new URL('https://example.org/?a=b&c=d');
myURL.searchParams.forEach((value, name, searchParams) => {
  console.log(name, value, myURL.searchParams === searchParams);
});
// Prints:
//   a b true
//   c d true
```

urlSearchParams.get(name)

- `name` <string>
- Returns: <string> or `null` if there is no name-value pair with the given `name`.

Returns the value of the first name-value pair whose name is `name`. If there are no such pairs, `null` is returned.

urlSearchParams.getAll(name)

- `name` <string>
- Returns: <Array>

Returns the values of all name-value pairs whose name is `name`. If there are no such pairs, an empty array is returned.

urlSearchParams.has(name)

- `name` <string>
- Returns: <boolean>

Returns `true` if there is at least one name-value pair whose name is `name`.

urlSearchParams.keys()

- Returns: <Iterator>

Returns an ES6 Iterator over the names of each name-value pair.

```
const { URLSearchParams } = require('url');
const params = new URLSearchParams('foo=bar&foo=baz');
for (const name of params.keys()) {
  console.log(name);
}
// Prints:
//   foo
//   foo
```

urlSearchParams.set(name, value)

- `name` <string>
- `value` <string>

Sets the value in the `URLSearchParams` object associated with `name` to `value`. If there are any pre-existing name-value pairs whose names are `name`, set the first such pair's value to `value` and remove all others. If not, append the name-value pair to the query string.

```js
const { URLSearchParams } = require('url');

const params = new URLSearchParams();
params.append('foo', 'bar');
params.append('foo', 'baz');
params.append('abc', 'def');
console.log(params.toString());
// Prints foo=bar&foo=baz&abc=def

params.set('foo', 'def');
params.set('xyz', 'opq');
console.log(params.toString());
// Prints foo=def&abc=def&xyz=opq
```

urlSearchParams.sort()

Added in: v7.7.0

Sort all existing name-value pairs in-place by their names. Sorting is done with a stable sorting algorithm, so relative order between name-value pairs with the same name is preserved.

This method can be used, in particular, to increase cache hits.

```js
const { URLSearchParams } = require('url');
const params = new URLSearchParams('query[]=abc&type=search&query[]=123');
params.sort();
console.log(params.toString());
// Prints query%5B%5D=abc&query%5B%5D=123&type=search
```

urlSearchParams.toString()

- Returns: <string>

Returns the search parameters serialized as a string, with characters percent-encoded where necessary.

urlSearchParams.values()

- Returns: <Iterator>

Returns an ES6 Iterator over the values of each name-value pair.

urlSearchParams[@@iterator]()

- Returns: <Iterator>

Returns an ES6 Iterator over each of the name-value pairs in the query string. Each item of the iterator is a JavaScript Array. The first item of the Array is the `name`, the second item of the Array is the `value`.

Alias for urlSearchParams.entries().

```js
const { URLSearchParams } = require('url');
const params = new URLSearchParams('foo=bar&xyz=baz');
for (const [name, value] of params) {
  console.log(name, value);
}
// Prints:
//   foo bar
//   xyz baz
```

url.domainToASCII(domain)

Added in: v7.4.0

- `domain` <string>
- Returns: <string>

Returns the Punycode ASCII serialization of the `domain`. If `domain` is an invalid domain, the empty string is returned.

It performs the inverse operation to url.domainToUnicode().

```
const url = require('url');
console.log(url.domainToASCII('español.com'));
// Prints xn--espaol-zwa.com
console.log(url.domainToASCII('□□.com'));
// Prints xn--fiq228c.com
console.log(url.domainToASCII('xn--iñvalid.com'));
// Prints an empty string
```

url.domainToUnicode(domain)

Added in: v7.4.0

- `domain` <string>
- Returns: <string>

Returns the Unicode serialization of the `domain`. If `domain` is an invalid domain, the empty string is returned.

It performs the inverse operation to url.domainToASCII().

```
const url = require('url');
console.log(url.domainToUnicode('xn--espaol-zwa.com'));
// Prints español.com
console.log(url.domainToUnicode('xn--fiq228c.com'));
// Prints □□.com
console.log(url.domainToUnicode('xn--iñvalid.com'));
// Prints an empty string
```

url.format(URL[, options])

Added in: v7.6.0

- `URL` <URL> A WHATWG URL object
- `options` <Object>
  - `auth` <boolean> `true` if the serialized URL string should include the username and password, `false` otherwise. Defaults to `true`.
  - `fragment` <boolean> `true` if the serialized URL string should include the fragment, `false` otherwise. Defaults to `true`.
  - `search` <boolean> `true` if the serialized URL string should include the search query, `false` otherwise. Defaults to `true`.
  - `unicode` <boolean> `true` if Unicode characters appearing in the host component of the URL string should be encoded directly as opposed to being Punycode encoded. Defaults to `false`.

Returns a customizable serialization of a URL String representation of a WHATWG URL object.

The URL object has both a `toString()` method and `href` property that return string serializations of the URL. These are not, however, customizable in any way. The `url.format(URL[, options])` method allows for basic customization of the output.

For example:

```
const { URL } = require('url');
const myURL = new URL('https://a:b@□□□□?abc#foo');

console.log(myURL.href);
// Prints https://a:b@xn--6qqa088eba/?abc#foo

console.log(myURL.toString());
// Prints https://a:b@xn--6qqa088eba/?abc#foo

console.log(url.format(myURL, { fragment: false, unicode: true, auth: false }));
// Prints 'https://□□□□/?abc'
```

Legacy URL API

Legacy urlObject

The legacy urlObject (`require('url').Url`) is created and returned by the `url.parse()` function.

urlObject.auth

The `auth` property is the username and password portion of the URL, also referred to as "userinfo". This string subset follows the `protocol` and double slashes (if present) and precedes the `host` component, delimited by an ASCII "at sign" (`@`). The format of the string is `{username}[:{password}]` , with the `[:{password}]` portion being optional.

For example: `'user:pass'`

urlObject.hash

The `hash` property consists of the "fragment" portion of the URL including the leading ASCII hash (`#`) character.

For example: `'#hash'`

urlObject.host

The `host` property is the full lower-cased host portion of the URL, including the `port` if specified.

For example: `'sub.host.com:8080'`

urlObject.hostname

The `hostname` property is the lower-cased host name portion of the `host` component *without* the `port` included.

For example: `'sub.host.com'`

urlObject.href

The `href` property is the full URL string that was parsed with both the `protocol` and `host` components converted to lower-case.

For example: `'http://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash'`

urlObject.path

The `path` property is a concatenation of the `pathname` and `search` components.

For example: `'/p/a/t/h?query=string'`

No decoding of the `path` is performed.

urlObject.pathname

The `pathname` property consists of the entire path section of the URL. This is everything following the `host` (including the `port`) and before the start of the `query` or `hash` components, delimited by either the ASCII question mark (`?`) or hash (`#`) characters.

For example `'/p/a/t/h'`

No decoding of the path string is performed.

urlObject.port

The `port` property is the numeric port portion of the `host` component.

For example: `'8080'`

urlObject.protocol

The `protocol` property identifies the URL's lower-cased protocol scheme.

For example: `'http:'`

urlObject.query

The `query` property is either the query string without the leading ASCII question mark (`?`), or an object returned by the querystring module's `parse()` method. Whether the `query` property is a string or object is determined by the `parseQueryString` argument passed to `url.parse()`.

For example: `'query=string'` or `{'query': 'string'}`

If returned as a string, no decoding of the query string is performed. If returned as an object, both keys and values are decoded.

urlObject.search

The `search` property consists of the entire "query string" portion of the URL, including the leading ASCII question mark (`?`) character.

For example: `'?query=string'`

No decoding of the query string is performed.

urlObject.slashes

The `slashes` property is a `boolean` with a value of `true` if two ASCII forward-slash characters (`/`) are required following the colon in the `protocol`.

url.format(urlObject)

▸ History

- `urlObject` <Object> | <string> A URL object (as returned by `url.parse()` or constructed otherwise). If a string, it is converted to an object by passing it to `url.parse()`.

The `url.format()` method returns a formatted URL string derived from `urlObject`.

```
url.format({
  protocol: 'https',
  hostname: 'example.com',
  pathname: '/some/path',
  query: {
    page: 1,
    format: 'json'
  }
});

// => 'https://example.com/some/path?page=1&format=json'
```

If `urlObject` is not an object or a string, `url.format()` will throw a TypeError.

The formatting process operates as follows:

- A new empty string `result` is created.
- If `urlObject.protocol` is a string, it is appended as-is to `result`.
- Otherwise, if `urlObject.protocol` is not `undefined` and is not a string, an Error is thrown.
- For all string values of `urlObject.protocol` that *do not end* with an ASCII colon (`:`) character, the literal string `:` will be appended to `result`.
- If either of the following conditions is true, then the literal string `//` will be appended to `result`:
  - `urlObject.slashes` property is true;
  - `urlObject.protocol` begins with `http`, `https`, `ftp`, `gopher`, or `file`;
- If the value of the `urlObject.auth` property is truthy, and either `urlObject.host` or `urlObject.hostname` are not `undefined`, the value of `urlObject.auth` will be coerced into a string and appended to `result` followed by the literal string `@`.
- If the `urlObject.host` property is `undefined` then:
  - If the `urlObject.hostname` is a string, it is appended to `result`.
  - Otherwise, if `urlObject.hostname` is not `undefined` and is not a string, an Error is thrown.
  - If the `urlObject.port` property value is truthy, and `urlObject.hostname` is not `undefined`:
    - The literal string `:` is appended to `result`, and
    - The value of `urlObject.port` is coerced to a string and appended to `result`.
- Otherwise, if the `urlObject.host` property value is truthy, the value of `urlObject.host` is coerced to a string and appended to `result`.
- If the `urlObject.pathname` property is a string that is not an empty string:
  - If the `urlObject.pathname` *does not start* with an ASCII forward slash (`/`), then the literal string '/' is appended to `result`.
  - The value of `urlObject.pathname` is appended to `result`.
- Otherwise, if `urlObject.pathname` is not `undefined` and is not a string, an Error is thrown.
- If the `urlObject.search` property is `undefined` and if the `urlObject.query` property is an `Object`, the literal string `?` is appended to `result` followed by the output of calling the querystring module's `stringify()` method passing the value of `urlObject.query`.
- Otherwise, if `urlObject.search` is a string:
  - If the value of `urlObject.search` *does not start* with the ASCII question mark (`?`) character, the literal string `?` is appended to `result`.
  - The value of `urlObject.search` is appended to `result`.
- Otherwise, if `urlObject.search` is not `undefined` and is not a string, an Error is thrown.
- If the `urlObject.hash` property is a string:
  - If the value of `urlObject.hash` *does not start* with the ASCII hash (`#`) character, the literal string `#` is appended to `result`.
  - The value of `urlObject.hash` is appended to `result`.
- Otherwise, if the `urlObject.hash` property is not `undefined` and is not a string, an Error is thrown.
- `result` is returned.

url.parse(urlString[, parseQueryString[, slashesDenoteHost]])

▸ History

- `urlString` <string> The URL string to parse.
- `parseQueryString` <boolean> If `true`, the `query` property will always be set to an object returned by the querystring module's `parse()` method. If `false`, the `query` property on the returned URL object will be an unparsed, undecoded string. Defaults to `false`.
- `slashesDenoteHost` <boolean> If `true`, the first token after the literal string `//` and preceding the next `/` will be interpreted as the `host`. For instance, given `//foo/bar`, the result would be `{host: 'foo', pathname: '/bar'}` rather than `{pathname: '//foo/bar'}`. Defaults to `false`.

The `url.parse()` method takes a URL string, parses it, and returns a URL object.

A `TypeError` is thrown if `urlString` is not a string.

A `URIError` is thrown if the `auth` property is present but cannot be decoded.

url.resolve(from, to)

▸ History

- `from` <string> The Base URL being resolved against.
- `to` <string> The HREF URL being resolved.

The `url.resolve()` method resolves a target URL relative to a base URL in a manner similar to that of a Web browser resolving an anchor tag HREF.

For example:

```
const url = require('url');
url.resolve('/one/two/three', 'four');         // '/one/two/four'
url.resolve('http://example.com/', '/one');    // 'http://example.com/one'
url.resolve('http://example.com/one', '/two'); // 'http://example.com/two'
```

Percent-Encoding in URLs

URLs are permitted to only contain a certain range of characters. Any character falling outside of that range must be encoded. How such characters are encoded, and which characters to encode depends entirely on where the character is located within the structure of the URL.

Legacy API

Within the Legacy API, spaces (`' '`) and the following characters will be automatically escaped in the properties of URL objects:

```
< > " ` \r \n \t { } | \ ^ '
```

For example, the ASCII space character (`' '`) is encoded as `%20`. The ASCII forward slash (`/`) character is encoded as `%3C`.

WHATWG API

The WHATWG URL Standard uses a more selective and fine grained approach to selecting encoded characters than that used by the Legacy API.

The WHATWG algorithm defines four "percent-encode sets" that describe ranges of characters that must be percent-encoded:

- The *C0 control percent-encode set* includes code points in range U+0000 to U+001F (inclusive) and all code points greater than U+007E.
- The *fragment percent-encode set* includes the *C0 control percent-encode set* and code points U+0020, U+0022, U+003C, U+003E, and U+0060.
- The *path percent-encode set* includes the *C0 control percent-encode set* and code points U+0020, U+0022, U+0023, U+003C, U+003E, U+003F, U+0060, U+007B, and U+007D.
- The *userinfo encode set* includes the *path percent-encode set* and code points U+002F, U+003A, U+003B, U+003D, U+0040, U+005B, U+005C, U+005D, U+005E, and U+007C.

The *userinfo percent-encode set* is used exclusively for username and passwords encoded within the URL. The *path percent-encode set* is used for the path of most URLs. The *fragment percent-encode set* is used for URL fragments. The *C0 control percent-encode set* is used for host and path under certain specific conditions, in addition to all other cases.

When non-ASCII characters appear within a hostname, the hostname is encoded using the Punycode algorithm. Note, however, that a hostname *may* contain *both* Punycode encoded and percent-encoded characters. For example:

```
const { URL } = require('url');
const myURL = new URL('https://%CF%80.com/foo');
console.log(myURL.href);
// Prints https://xn--1xa.com/foo
console.log(myURL.origin);
// Prints https://π.com
```

Util

Stability: 2 - Stable

The `util` module is primarily designed to support the needs of Node.js' own internal APIs. However, many of the utilities are useful for application and module developers as well. It can be accessed using:

```
const util = require('util');
```

util.callbackify(original)

Added in: v8.2.0

- `original` <Function> An `async` function
- Returns: <Function> a callback style function

Takes an `async` function (or a function that returns a Promise) and returns a function following the error-first callback style, i.e. taking a `(err, value) => ...` callback as the last argument. In the callback, the first argument will be the rejection reason (or `null` if the Promise resolved), and the second argument will be the resolved value.

```
const util = require('util');

async function fn() {
  return 'hello world';
}
const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  if (err) throw err;
  console.log(ret);
});
```

Will print:

```
hello world
```

The callback is executed asynchronously, and will have a limited stack trace. If the callback throws, the process will emit an `'uncaughtException'` event, and if not handled will exit.

Since `null` has a special meaning as the first argument to a callback, if a wrapped function rejects a `Promise` with a falsy value as a reason, the value is wrapped in an `Error` with the original value stored in a field named `reason`.

```
function fn() {
  return Promise.reject(null);
}
const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  // When the Promise was rejected with `null` it is wrapped with an Error and
  // the original value is stored in `reason`.
  err && err.hasOwnProperty('reason') && err.reason === null;  // true
});
```

util.debuglog(section)

Added in: v0.11.3

- `section` <string> A string identifying the portion of the application for which the `debuglog` function is being created.
- Returns: <Function> The logging function

The `util.debuglog()` method is used to create a function that conditionally writes debug messages to `stderr` based on the existence of the `NODE_DEBUG` environment variable. If the `section` name appears within the value of that environment variable, then the returned function operates similar to `console.error()`. If not, then the returned function is a no-op.

```
const util = require('util');
const debuglog = util.debuglog('foo');

debuglog('hello from foo [%d]', 123);
```

If this program is run with `NODE_DEBUG=foo` in the environment, then it will output something like:

```
FOO 3245: hello from foo [123]
```

where `3245` is the process id. If it is not run with that environment variable set, then it will not print anything.

The `section` supports wildcard also:

```
const util = require('util');
const debuglog = util.debuglog('foo-bar');

debuglog('hi there, it\'s foo-bar [%d]', 2333);
```

if it is run with `NODE_DEBUG=foo*` in the environment, then it will output something like:

```
FOO-BAR 3257: hi there, it's foo-bar [2333]
```

Multiple comma-separated `section` names may be specified in the `NODE_DEBUG` environment variable: `NODE_DEBUG=fs,net,tls`.

util.deprecate(function, string)

Added in: v0.8.0

The `util.deprecate()` method wraps the given `function` or class in such a way that it is marked as deprecated.

```
const util = require('util');

exports.obsoleteFunction = util.deprecate(() => {
  // Do something here.
}, 'obsoleteFunction() is deprecated. Use newShinyFunction() instead.');
```

When called, `util.deprecate()` will return a function that will emit a `DeprecationWarning` using the `process.on('warning')` event. By default, this warning will be emitted and printed to `stderr` exactly once, the first time it is called. After the warning is emitted, the wrapped `function` is called.

If either the `--no-deprecation` or `--no-warnings` command line flags are used, or if the `process.noDeprecation` property is set to `true` *prior* to the first deprecation warning, the `util.deprecate()` method does nothing.

If the `--trace-deprecation` or `--trace-warnings` command line flags are set, or the `process.traceDeprecation` property is set to `true`, a warning and a stack trace are printed to `stderr` the first time the deprecated function is called.

If the `--throw-deprecation` command line flag is set, or the `process.throwDeprecation` property is set to `true`, then an exception will be thrown when the deprecated function is called.

The `--throw-deprecation` command line flag and `process.throwDeprecation` property take precedence over `--trace-deprecation` and `process.traceDeprecation`.

util.format(format[, ...args])

▸ History

- `format` <string> A `printf`-like format string.

The `util.format()` method returns a formatted string using the first argument as a `printf`-like format.

The first argument is a string containing zero or more *placeholder* tokens. Each placeholder token is replaced with the converted value from the corresponding argument. Supported placeholders are:

- `%s` - String.
- `%d` - Number (integer or floating point value).
- `%i` - Integer.
- `%f` - Floating point value.
- `%j` - JSON. Replaced with the string `'[Circular]'` if the argument contains circular references.
- `%o` - Object. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` with options `{ showHidden: true, depth: 4, showProxy: true }`. This will show the full object including non-enumerable symbols and properties.
- `%O` - Object. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` without options. This will show the full object not including non-enumerable symbols and properties.
- `%%` - single percent sign (`'%'`). This does not consume an argument.
- Returns: <string> The formatted string

If the placeholder does not have a corresponding argument, the placeholder is not replaced.

```
util.format('%s:%s', 'foo');
// Returns: 'foo:%s'
```

If there are more arguments passed to the `util.format()` method than the number of placeholders, the extra arguments are coerced into strings then concatenated to the returned string, each delimited by a space. Excessive arguments whose `typeof` is `'object'` or `'symbol'` (except `null`) will be transformed by `util.inspect()`.

```
util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

If the first argument is not a string then `util.format()` returns a string that is the concatenation of all arguments separated by spaces. Each argument is converted to a string using `util.inspect()`.

```
util.format(1, 2, 3); // '1 2 3'
```

If only one argument is passed to `util.format()`, it is returned as it is without any formatting.

```
util.format('%% %s'); // '%% %s'
```

Please note that `util.format()` is a synchronous method that is mainly intended as a debugging tool. Some input values can have a significant performance overhead that can block the event loop. Use this function with care and never in a hot code path.

util.getSystemErrorName(err)

Added in: v9.7.0

- `err` <number>
- Returns: <string>

Returns the string name for a numeric error code that comes from a Node.js API. The mapping between error codes and error names is platform-dependent. See Common System Errors for the names of common errors.

```
fs.access('file/that/does/not/exist', (err) => {
  const name = util.getSystemErrorName(err.errno);
  console.error(name);  // ENOENT
});
```

util.inherits(constructor, superConstructor)

▶ History

Usage of `util.inherits()` is discouraged. Please use the ES6 `class` and `extends` keywords to get language level inheritance support. Also note that the two styles are semantically incompatible.

- `constructor` <Function>
- `superConstructor` <Function>

Inherit the prototype methods from one constructor into another. The prototype of `constructor` will be set to a new object created from `superConstructor`.

As an additional convenience, `superConstructor` will be accessible through the `constructor.super_` property.

```
const util = require('util');
const EventEmitter = require('events');

function MyStream() {
  EventEmitter.call(this);
}

util.inherits(MyStream, EventEmitter);

MyStream.prototype.write = function(data) {
  this.emit('data', data);
};

const stream = new MyStream();

console.log(stream instanceof EventEmitter); // true
console.log(MyStream.super_ === EventEmitter); // true

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('It works!'); // Received data: "It works!"
```

ES6 example using `class` and `extends`

```
const EventEmitter = require('events');

class MyStream extends EventEmitter {
  write(data) {
    this.emit('data', data);
  }
}

const stream = new MyStream();

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('With ES6');
```

util.inspect(object[, options])

▸ History

- `object` <any> Any JavaScript primitive or Object.
- `options` <Object>
  - `showHidden` <boolean> If `true`, the `object`'s non-enumerable symbols and properties will be included in the formatted result. Defaults to `false`.
  - `depth` <number> Specifies the number of times to recurse while formatting the `object`. This is useful for inspecting large complicated objects. Defaults to `2`. To make it recurse indefinitely pass `null`.
  - `colors` <boolean> If `true`, the output will be styled with ANSI color codes. Defaults to `false`. Colors are customizable, see Customizing `util.inspect` colors.
  - `customInspect` <boolean> If `false`, then custom `inspect(depth, opts)` functions will not be called. Defaults to `true`.
  - `showProxy` <boolean> If `true`, then objects and functions that are `Proxy` objects will be introspected to show their `target` and `handler`

objects. Defaults to `false`.

- `maxArrayLength` <number> Specifies the maximum number of array and `TypedArray` elements to include when formatting. Defaults to `100`. Set to `null` to show all array elements. Set to `0` or negative to show no array elements.
- `breakLength` <number> The length at which an object's keys are split across multiple lines. Set to `Infinity` to format an object as a single line. Defaults to 60 for legacy compatibility.
- `compact` <boolean> Setting this to `false` changes the default indentation to use a line break for each object key instead of lining up multiple properties in one line. It will also break text that is above the `breakLength` size into smaller and better readable chunks and indents objects the same as arrays. Note that no text will be reduced below 16 characters, no matter the `breakLength` size. For more information, see the example below. Defaults to `true`.

The `util.inspect()` method returns a string representation of `object` that is primarily useful for debugging. Additional `options` may be passed that alter certain aspects of the formatted string.

The following example inspects all properties of the `util` object:

```
const util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));
```

Values may supply their own custom `inspect(depth, opts)` functions, when called these receive the current `depth` in the recursive inspection, as well as the options object passed to `util.inspect()`.

The following example highlights the difference with the `compact` option:

```
const util = require('util');

const o = {
  a: [1, 2, [[
    'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do ' +
      'eiusmod tempor incididunt ut labore et dolore magna aliqua.',
    'test',
    'foo']], 4],
  b: new Map([['za', 1], ['zb', 'test']])
};
console.log(util.inspect(o, { compact: true, depth: 5, breakLength: 80 }));

// This will print

// { a:
//   [ 1,
//     2,
//     [ [ 'Lorem ipsum dolor sit amet, consectetur [...]', // A long line
//           'test',
//           'foo' ] ],
//     4 ],
//   b: Map { 'za' => 1, 'zb' => 'test' } }

// Setting `compact` to false changes the output to be more reader friendly.
console.log(util.inspect(o, { compact: false, depth: 5, breakLength: 80 }));

// {
//   a: [
//     1,
//     2,
//     [
//       [
//         'Lorem ipsum dolor sit amet, consectetur ' +
//           'adipiscing elit, sed do eiusmod tempor ' +
//           'incididunt ut labore et dolore magna ' +
//           'aliqua.,
//         'test',
//         'foo'
//       ]
//     ],
//     4
//   ],
//   b: Map {
//     'za' => 1,
//     'zb' => 'test'
//   }
// }

// Setting `breakLength` to e.g. 150 will print the "Lorem ipsum" text in a
// single line.
// Reducing the `breakLength` will split the "Lorem ipsum" text in smaller
// chunks.
```

Please note that `util.inspect()` is a synchronous method that is mainly intended as a debugging tool. Some input values can have a significant performance overhead that can block the event loop. Use this function with care and never in a hot code path.

Customizing `util.inspect` colors

Color output (if enabled) of `util.inspect` is customizable globally via the `util.inspect.styles` and `util.inspect.colors` properties.

`util.inspect.styles` is a map associating a style name to a color from `util.inspect.colors`.

The default styles and associated colors are:

- `number` - `yellow`
- `boolean` - `yellow`
- `string` - `green`
- `date` - `magenta`
- `regexp` - `red`
- `null` - `bold`
- `undefined` - `grey`
- `special` - `cyan` (only applied to functions at this time)
- `name` - (no styling)

The predefined color codes are: `white`, `grey`, `black`, `blue`, `cyan`, `green`, `magenta`, `red` and `yellow`. There are also `bold`, `italic`, `underline` and `inverse` codes.

Color styling uses ANSI control codes that may not be supported on all terminals.

Custom inspection functions on Objects

Objects may also define their own `[util.inspect.custom](depth, opts)` (or the equivalent but deprecated `inspect(depth, opts)`) function that `util.inspect()` will invoke and use the result of when inspecting the object:

```
const util = require('util');

class Box {
  constructor(value) {
    this.value = value;
  }

  [util.inspect.custom](depth, options) {
    if (depth < 0) {
      return options.stylize('[Box]', 'special');
    }

    const newOptions = Object.assign({}, options, {
      depth: options.depth === null ? null : options.depth - 1
    });

    // Five space padding because that's the size of "Box< ".
    const padding = ' '.repeat(5);
    const inner = util.inspect(this.value, newOptions)
                  .replace(/\n/g, `\n${padding}`);
    return `${options.stylize('Box', 'special')}< ${inner} >`;
  }
}

const box = new Box(true);

util.inspect(box);
// Returns: "Box< true >"
```

Custom [util.inspect.custom](depth, opts) functions typically return a string but may return a value of any type that will be formatted accordingly by util.inspect().

```
const util = require('util');

const obj = { foo: 'this will not show up in the inspect() output' };
obj[util.inspect.custom] = (depth) => {
  return { bar: 'baz' };
};

util.inspect(obj);
// Returns: "{ bar: 'baz' }"
```

util.inspect.custom

Added in: v6.6.0

A Symbol that can be used to declare custom inspect functions, see Custom inspection functions on Objects.

util.inspect.defaultOptions

Added in: v6.4.0

The defaultOptions value allows customization of the default options used by util.inspect. This is useful for functions like console.log or util.format which implicitly call into util.inspect. It shall be set to an object containing one or more valid util.inspect() options. Setting option properties directly is also supported.

```
const util = require('util');
const arr = Array(101).fill(0);

console.log(arr); // logs the truncated array
util.inspect.defaultOptions.maxArrayLength = null;
console.log(arr); // logs the full array
```

util.isDeepStrictEqual(val1, val2)

Added in: v9.0.0

- val1 <any>
- val2 <any>
- Returns: <boolean>

Returns true if there is deep strict equality between val1 and val2. Otherwise, returns false.

See assert.deepStrictEqual() for more information about deep strict equality.

util.promisify(original)

Added in: v8.0.0

- original <Function>
- Returns: <Function>

Takes a function following the common error-first callback style, i.e. taking a (err, value) => ... callback as the last argument, and returns a version that returns promises.

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```

Or, equivalently using `async function` s:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

async function callStat() {
  const stats = await stat('.');
  console.log(`This directory is owned by ${stats.uid}`);
}
```

If there is an `original[util.promisify.custom]` property present, `promisify` will return its value, see Custom promisified functions .

`promisify()` assumes that `original` is a function taking a callback as its final argument in all cases. If `original` is not a function, `promisify()` will throw an error. If `original` is a function but its last argument is not an error-first callback, it will still be passed an error-first callback as its last argument.

Custom promisified functions

Using the `util.promisify.custom` symbol one can override the return value of util.promisify() :

```
const util = require('util');

function doSomething(foo, callback) {
  // ...
}

doSomething[util.promisify.custom] = (foo) => {
  return getPromiseSomehow();
};

const promisified = util.promisify(doSomething);
console.log(promisified === doSomething[util.promisify.custom]);
// prints 'true'
```

This can be useful for cases where the original function does not follow the standard format of taking an error-first callback as the last argument.

For example, with a function that takes in `(foo, onSuccessCallback, onErrorCallback)` :

```
doSomething[util.promisify.custom] = (foo) => {
  return new Promise((resolve, reject) => {
    doSomething(foo, resolve, reject);
  });
};
```

If `promisify.custom` is defined but is not a function, `promisify()` will throw an error.

util.promisify.custom

Added in: v8.0.0

- <symbol>

A Symbol that can be used to declare custom promisified variants of functions, see Custom promisified functions .

Class: util.TextDecoder

Added in: v8.3.0

An implementation of the WHATWG Encoding Standard `TextDecoder` API.

```
const decoder = new TextDecoder('shift_jis');
let string = '';
let buffer;
while (buffer = getNextChunkSomehow()) {
  string += decoder.decode(buffer, { stream: true });
}
string += decoder.decode(); // end-of-stream
```

WHATWG Supported Encodings

Per the WHATWG Encoding Standard , the encodings supported by the `TextDecoder` API are outlined in the tables below. For each encoding, one or more aliases may be used.

Different Node.js build configurations support different sets of encodings. While a very basic set of encodings is supported even on Node.js builds without ICU enabled, support for some encodings is provided only when Node.js is built with ICU and using the full ICU data (see Internationalization).

Encodings Supported Without ICU

Encoding	Aliases
`'utf-8'`	`'unicode-1-1-utf-8'`, `'utf8'`
`'utf-16le'`	`'utf-16'`

Encodings Supported by Default (With ICU)

Encoding	Aliases
`'utf-8'`	`'unicode-1-1-utf-8'`, `'utf8'`
`'utf-16le'`	`'utf-16'`
`'utf-16be'`	

Encodings Requiring Full ICU Data

Encoding	Aliases
'ibm866'	'866', 'cp866', 'csibm866'
'iso-8859-2'	'csisolatin2', 'iso-ir-101', 'iso8859-2', 'iso88592', 'iso_8859-2', 'iso_8859-2:1987', 'l2', 'latin2'
'iso-8859-3'	'csisolatin3', 'iso-ir-109', 'iso8859-3', 'iso88593', 'iso_8859-3', 'iso_8859-3:1988', 'l3', 'latin3'
'iso-8859-4'	'csisolatin4', 'iso-ir-110', 'iso8859-4', 'iso88594', 'iso_8859-4', 'iso_8859-4:1988', 'l4', 'latin4'
'iso-8859-5'	'csisolatincyrillic', 'cyrillic', 'iso-ir-144', 'iso8859-5', 'iso88595', 'iso_8859-5', 'iso_8859-5:1988'
'iso-8859-6'	'arabic', 'asmo-708', 'csiso88596e', 'csiso88596i', 'csisolatinarabic', 'ecma-114', 'iso-8859-6-e', 'iso-8859-6-i', 'iso-ir-127', 'iso8859-6', 'iso88596', 'iso_8859-6', 'iso_8859-6:1987'
'iso-8859-7'	'csisolatingreek', 'ecma-118', 'elot_928', 'greek', 'greek8', 'iso-ir-126', 'iso8859-7', 'iso88597', 'iso_8859-7', 'iso_8859-7:1987', 'sun_eu_greek'
'iso-8859-8'	'csiso88598e', 'csisolatinhebrew', 'hebrew', 'iso-8859-8-e', 'iso-ir-138', 'iso8859-8', 'iso88598', 'iso_8859-8', 'iso_8859-8:1988', 'visual'
'iso-8859-8-i'	'csiso88598i', 'logical'
'iso-8859-10'	'csisolatin6', 'iso-ir-157', 'iso8859-10', 'iso885910', 'l6', 'latin6'
'iso-8859-13'	'iso8859-13', 'iso885913'
'iso-8859-14'	'iso8859-14', 'iso885914'
'iso-8859-15'	'csisolatin9', 'iso8859-15', 'iso885915', 'iso_8859-15', 'l9'
'koi8-r'	'cskoi8r', 'koi', 'koi8', 'koi8_r'
'koi8-u'	'koi8-ru'
'macintosh'	'csmacintosh', 'mac', 'x-mac-roman'
'windows-874'	'dos-874', 'iso-8859-11', 'iso8859-11', 'iso885911', 'tis-620'
'windows-1250'	'cp1250', 'x-cp1250'
'windows-1251'	'cp1251', 'x-cp1251'
'windows-1252'	'ansi_x3.4-1968', 'ascii', 'cp1252', 'cp819', 'csisolatin1', 'ibm819', 'iso-8859-1', 'iso-ir-100', 'iso8859-1', 'iso88591', 'iso_8859-1', 'iso_8859-1:1987', 'l1', 'latin1', 'us-ascii', 'x-cp1252'

Encoding	Aliases
'windows-1253'	'cp1253' , 'x-cp1253'
'windows-1254'	'cp1254' , 'csisolatin5' , 'iso-8859-9' , 'iso-ir-148' , 'iso8859-9' , 'iso88599' , 'iso_8859-9' , 'iso_8859-9:1989' , 'l5' , 'latin5' , 'x-cp1254'
'windows-1255'	'cp1255' , 'x-cp1255'
'windows-1256'	'cp1256' , 'x-cp1256'
'windows-1257'	'cp1257' , 'x-cp1257'
'windows-1258'	'cp1258' , 'x-cp1258'
'x-mac-cyrillic'	'x-mac-ukrainian'
'gbk'	'chinese' , 'csgb2312' , 'csiso58gb231280' , 'gb2312' , 'gb_2312' , 'gb_2312-80' , 'iso-ir-58' , 'x-gbk'
'gb18030'	
'big5'	'big5-hkscs' , 'cn-big5' , 'csbig5' , 'x-x-big5'
'euc-jp'	'cseucpkdfmtjapanese' , 'x-euc-jp'
'iso-2022-jp'	'csiso2022jp'
'shift_jis'	'csshiftjis' , 'ms932' , 'ms_kanji' , 'shift-jis' , 'sjis' , 'windows-31j' , 'x-sjis'
'euc-kr'	'cseuckr' , 'csksc56011987' , 'iso-ir-149' , 'korean' , 'ks_c_5601-1987' , 'ks_c_5601-1989' , 'ksc5601' , 'ksc_5601' , 'windows-949'

The 'iso-8859-16' encoding listed in the WHATWG Encoding Standard is not supported.

new TextDecoder([encoding[, options]])

- encoding <string> Identifies the encoding that this TextDecoder instance supports. Defaults to 'utf-8' .
- options <Object>
  - fatal <boolean> true if decoding failures are fatal. Defaults to false . This option is only supported when ICU is enabled (see Internationalization).
  - ignoreBOM <boolean> When true , the TextDecoder will include the byte order mark in the decoded result. When false , the byte order mark will be removed from the output. This option is only used when encoding is 'utf-8' , 'utf-16be' or 'utf-16le' . Defaults to false .

Creates an new TextDecoder instance. The encoding may specify one of the supported encodings or an alias.

textDecoder.decode([input[, options]])

- input <ArrayBuffer> | <DataView> | <TypedArray> An ArrayBuffer , DataView or Typed Array instance containing the encoded data.
- options <Object>
  - stream <boolean> true if additional chunks of data are expected. Defaults to false .
- Returns: <string>

Decodes the input and returns a string. If options.stream is true , any incomplete byte sequences occurring at the end of the input are buffered internally and emitted after the next call to textDecoder.decode() .

If `textDecoder.fatal` is `true`, decoding errors that occur will result in a `TypeError` being thrown.

textDecoder.encoding

- <string>

The encoding supported by the `TextDecoder` instance.

textDecoder.fatal

- <boolean>

The value will be `true` if decoding errors result in a `TypeError` being thrown.

textDecoder.ignoreBOM

- <boolean>

The value will be `true` if the decoding result will include the byte order mark.

Class: util.TextEncoder

Added in: v8.3.0

An implementation of the WHATWG Encoding Standard `TextEncoder` API. All instances of `TextEncoder` only support UTF-8 encoding.

```
const encoder = new TextEncoder();
const uint8array = encoder.encode('this is some data');
```

textEncoder.encode([input])

- `input` <string> The text to encode. Defaults to an empty string.
- Returns: <Uint8Array>

UTF-8 encodes the `input` string and returns a `Uint8Array` containing the encoded bytes.

textEncoder.encoding

- <string>

The encoding supported by the `TextEncoder` instance. Always set to `'utf-8'`.

Deprecated APIs

The following APIs have been deprecated and should no longer be used. Existing applications and modules should be updated to find alternative approaches.

util._extend(target, source)

Added in: v0.7.5 Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use `Object.assign()` instead.

The `util._extend()` method was never intended to be used outside of internal Node.js modules. The community found and used it anyway.

It is deprecated and should not be used in new code. JavaScript comes with very similar built-in functionality through `Object.assign()`.

util.debug(string)

Added in: v0.3.0 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use console.error() instead.

- `string` <string> The message to print to `stderr`

Deprecated predecessor of `console.error`.

util.error([...strings])

Added in: v0.3.0 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use console.error() instead.

- `...strings` <string> The message to print to `stderr`

Deprecated predecessor of `console.error`.

util.isArray(object)

Added in: v0.6.0 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Internal alias for Array.isArray.

Returns `true` if the given `object` is an `Array`. Otherwise, returns `false`.

```
const util = require('util');

util.isArray([]);
// Returns: true
util.isArray(new Array());
// Returns: true
util.isArray({});
// Returns: false
```

util.isBoolean(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a `Boolean`. Otherwise, returns `false`.

```
const util = require('util');

util.isBoolean(1);
// Returns: false
util.isBoolean(0);
// Returns: false
util.isBoolean(false);
// Returns: true
```

util.isBuffer(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated: Use Buffer.isBuffer() instead.

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a `Buffer`. Otherwise, returns `false`.

```
const util = require('util');

util.isBuffer({ length: 0 });
// Returns: false
util.isBuffer([]);
// Returns: false
util.isBuffer(Buffer.from('hello world'));
// Returns: true
```

util.isDate(object)

Added in: v0.6.0 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a `Date`. Otherwise, returns `false`.

```
const util = require('util');

util.isDate(new Date());
// Returns: true
util.isDate(Date());
// false (without 'new' returns a String)
util.isDate({});
// Returns: false
```

util.isError(object)

Added in: v0.6.0 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is an `Error`. Otherwise, returns `false`.

```
const util = require('util');

util.isError(new Error());
// Returns: true
util.isError(new TypeError());
// Returns: true
util.isError({ name: 'Error', message: 'an error occurred' });
// Returns: false
```

Note that this method relies on `Object.prototype.toString()` behavior. It is possible to obtain an incorrect result when the `object` argument manipulates `@@toStringTag`.

```
const util = require('util');
const obj = { name: 'Error', message: 'an error occurred' };

util.isError(obj);
// Returns: false
obj[Symbol.toStringTag] = 'Error';
util.isError(obj);
// Returns: true
```

util.isFunction(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a `Function`. Otherwise, returns `false`.

```
const util = require('util');

function Foo() {}
const Bar = () => {};

util.isFunction({});
// Returns: false
util.isFunction(Foo);
// Returns: true
util.isFunction(Bar);
// Returns: true
```

util.isNull(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- object <any>
- Returns: <boolean>

Returns true if the given object is strictly null. Otherwise, returns false.

```
const util = require('util');

util.isNull(0);
// Returns: false
util.isNull(undefined);
// Returns: false
util.isNull(null);
// Returns: true
```

util.isNullOrUndefined(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- object <any>
- Returns: <boolean>

Returns true if the given object is null or undefined. Otherwise, returns false.

```
const util = require('util');

util.isNullOrUndefined(0);
// Returns: false
util.isNullOrUndefined(undefined);
// Returns: true
util.isNullOrUndefined(null);
// Returns: true
```

util.isNumber(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- object <any>
- Returns: <boolean>

Returns true if the given object is a Number. Otherwise, returns false.

```
const util = require('util');

util.isNumber(false);
// Returns: false
util.isNumber(Infinity);
// Returns: true
util.isNumber(0);
// Returns: true
util.isNumber(NaN);
// Returns: true
```

util.isObject(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is strictly an `Object` **and** not a `Function`. Otherwise, returns `false`.

```
const util = require('util');

util.isObject(5);
// Returns: false
util.isObject(null);
// Returns: false
util.isObject({});
// Returns: true
util.isObject(() => {});
// Returns: false
```

util.isPrimitive(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a primitive type. Otherwise, returns `false`.

```
const util = require('util');

util.isPrimitive(5);
// Returns: true
util.isPrimitive('foo');
// Returns: true
util.isPrimitive(false);
// Returns: true
util.isPrimitive(null);
// Returns: true
util.isPrimitive(undefined);
// Returns: true
util.isPrimitive({});
// Returns: false
util.isPrimitive(() => {});
// Returns: false
util.isPrimitive(/^$/);
// Returns: false
util.isPrimitive(new Date());
// Returns: false
```

util.isRegExp(object)

Added in: v0.6.0   Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a `RegExp`. Otherwise, returns `false`.

```
const util = require('util');

util.isRegExp(/some regexp/);
// Returns: true
util.isRegExp(new RegExp('another regexp'));
// Returns: true
util.isRegExp({});
// Returns: false
```

util.isString(object)

Added in: v0.11.5   Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a `string`. Otherwise, returns `false`.

```
const util = require('util');

util.isString('');
// Returns: true
util.isString('foo');
// Returns: true
util.isString(String('foo'));
// Returns: true
util.isString(5);
// Returns: false
```

util.isSymbol(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is a `Symbol`. Otherwise, returns `false`.

```
const util = require('util');

util.isSymbol(5);
// Returns: false
util.isSymbol('foo');
// Returns: false
util.isSymbol(Symbol('foo'));
// Returns: true
```

util.isUndefined(object)

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>
- Returns: <boolean>

Returns `true` if the given `object` is `undefined`. Otherwise, returns `false`.

```
const util = require('util');

const foo = undefined;
util.isUndefined(5);
// Returns: false
util.isUndefined(foo);
// Returns: true
util.isUndefined(null);
// Returns: false
```

util.log(string)

Added in: v0.3.0 Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use a third party module instead.

- string <string>

The util.log() method prints the given string to stdout with an included timestamp.

```
const util = require('util');

util.log('Timestamped message.');
```

util.print([...strings])

Added in: v0.3.0 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use console.log() instead.

Deprecated predecessor of console.log.

util.puts([...strings])

Added in: v0.3.0 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use console.log() instead.

Deprecated predecessor of console.log.

V8

The v8 module exposes APIs that are specific to the version of V8 built into the Node.js binary. It can be accessed using:

```
const v8 = require('v8');
```

The APIs and implementation are subject to change at any time.

v8.cachedDataVersionTag()

Added in: v8.0.0

Returns an integer representing a "version tag" derived from the V8 version, command line flags and detected CPU features. This is useful for determining whether a vm.Script cachedData buffer is compatible with this instance of V8.

v8.getHeapSpaceStatistics()

▸ History

Returns statistics about the V8 heap spaces, i.e. the segments which make up the V8 heap. Neither the ordering of heap spaces, nor the availability of a heap space can be guaranteed as the statistics are provided via the V8 GetHeapSpaceStatistics function and may change from one V8 version to the next.

The value returned is an array of objects containing the following properties:

- space_name <string>
- space_size <number>
- space_used_size <number>

- `space_available_size` <number>
- `physical_space_size` <number>

```
[
  {
    "space_name": "new_space",
    "space_size": 2063872,
    "space_used_size": 951112,
    "space_available_size": 80824,
    "physical_space_size": 2063872
  },
  {
    "space_name": "old_space",
    "space_size": 3090560,
    "space_used_size": 2493792,
    "space_available_size": 0,
    "physical_space_size": 3090560
  },
  {
    "space_name": "code_space",
    "space_size": 1260160,
    "space_used_size": 644256,
    "space_available_size": 960,
    "physical_space_size": 1260160
  },
  {
    "space_name": "map_space",
    "space_size": 1094160,
    "space_used_size": 201608,
    "space_available_size": 0,
    "physical_space_size": 1094160
  },
  {
    "space_name": "large_object_space",
    "space_size": 0,
    "space_used_size": 0,
    "space_available_size": 1490980608,
    "physical_space_size": 0
  }
]
```

v8.getHeapStatistics()

▶ History

Returns an object with the following properties:

- `total_heap_size` <number>
- `total_heap_size_executable` <number>
- `total_physical_size` <number>
- `total_available_size` <number>
- `used_heap_size` <number>
- `heap_size_limit` <number>
- `malloced_memory` <number>
- `peak_malloced_memory` <number>

- does_zap_garbage <number>

does_zap_garbage is a 0/1 boolean, which signifies whether the --zap_code_space option is enabled or not. This makes V8 overwrite heap garbage with a bit pattern. The RSS footprint (resident memory set) gets bigger because it continuously touches all heap pages and that makes them less likely to get swapped out by the operating system.

```
{
  total_heap_size: 7326976,
  total_heap_size_executable: 4194304,
  total_physical_size: 7326976,
  total_available_size: 1152656,
  used_heap_size: 3476208,
  heap_size_limit: 1535115264,
  malloced_memory: 16384,
  peak_malloced_memory: 1127496,
  does_zap_garbage: 0
}
```

v8.setFlagsFromString(flags)

Added in: v1.0.0

- flags <string>

The v8.setFlagsFromString() method can be used to programmatically set V8 command line flags. This method should be used with care. Changing settings after the VM has started may result in unpredictable behavior, including crashes and data loss; or it may simply do nothing.

The V8 options available for a version of Node.js may be determined by running node --v8-options . An unofficial, community-maintained list of options and their effects is available here .

Usage:

```
// Print GC events to stdout for one minute.
const v8 = require('v8');
v8.setFlagsFromString('--trace_gc');
setTimeout(function() { v8.setFlagsFromString('--notrace_gc'); }, 60e3);
```

Serialization API

Stability: 1 - Experimental

The serialization API provides means of serializing JavaScript values in a way that is compatible with the HTML structured clone algorithm . The format is backward-compatible (i.e. safe to store to disk).

This API is under development, and changes (including incompatible changes to the API or wire format) may occur until this warning is removed.

v8.serialize(value)

- Returns: <Buffer>

Uses a DefaultSerializer to serialize value into a buffer.

v8.deserialize(buffer)

- buffer <Buffer> | <Uint8Array> A buffer returned by serialize() .

Uses a DefaultDeserializer with default options to read a JS value from a buffer.

class: v8.Serializer

new Serializer()

Creates a new `Serializer` object.

serializer.writeHeader()

Writes out a header, which includes the serialization format version.

serializer.writeValue(value)

Serializes a JavaScript value and adds the serialized representation to the internal buffer.

This throws an error if `value` cannot be serialized.

serializer.releaseBuffer()

Returns the stored internal buffer. This serializer should not be used once the buffer is released. Calling this method results in undefined behavior if a previous write has failed.

serializer.transferArrayBuffer(id, arrayBuffer)

- `id` <integer> A 32-bit unsigned integer.
- `arrayBuffer` <ArrayBuffer> An `ArrayBuffer` instance.

Marks an `ArrayBuffer` as havings its contents transferred out of band. Pass the corresponding `ArrayBuffer` in the deserializing context to deserializer.transferArrayBuffer() .

serializer.writeUint32(value)

- `value` <integer>

Write a raw 32-bit unsigned integer. For use inside of a custom serializer._writeHostObject() .

serializer.writeUint64(hi, lo)

- `hi` <integer>
- `lo` <integer>

Write a raw 64-bit unsigned integer, split into high and low 32-bit parts. For use inside of a custom serializer._writeHostObject() .

serializer.writeDouble(value)

- `value` <number>

Write a JS `number` value. For use inside of a custom serializer._writeHostObject() .

serializer.writeRawBytes(buffer)

- `buffer` <Buffer> | <Uint8Array>

Write raw bytes into the serializer's internal buffer. The deserializer will require a way to compute the length of the buffer. For use inside of a custom serializer._writeHostObject() .

serializer._writeHostObject(object)

- `object` <Object>

This method is called to write some kind of host object, i.e. an object created by native C++ bindings. If it is not possible to serialize `object`, a suitable exception should be thrown.

This method is not present on the `Serializer` class itself but can be provided by subclasses.

serializer._getDataCloneError(message)

- message <string>

This method is called to generate error objects that will be thrown when an object can not be cloned.

This method defaults to the Error constructor and can be overridden on subclasses.

serializer._getSharedArrayBufferId(sharedArrayBuffer)

- sharedArrayBuffer <SharedArrayBuffer>

This method is called when the serializer is going to serialize a SharedArrayBuffer object. It must return an unsigned 32-bit integer ID for the object, using the same ID if this SharedArrayBuffer has already been serialized. When deserializing, this ID will be passed to deserializer.transferArrayBuffer() .

If the object cannot be serialized, an exception should be thrown.

This method is not present on the Serializer class itself but can be provided by subclasses.

serializer._setTreatArrayBufferViewsAsHostObjects(flag)

- flag <boolean>

Indicate whether to treat TypedArray and DataView objects as host objects, i.e. pass them to serializer._writeHostObject() .

The default is not to treat those objects as host objects.

class: v8.Deserializer

new Deserializer(buffer)

- buffer <Buffer> | <Uint8Array> A buffer returned by serializer.releaseBuffer() .

Creates a new Deserializer object.

deserializer.readHeader()

Reads and validates a header (including the format version). May, for example, reject an invalid or unsupported wire format. In that case, an Error is thrown.

deserializer.readValue()

Deserializes a JavaScript value from the buffer and returns it.

deserializer.transferArrayBuffer(id, arrayBuffer)

- id <integer> A 32-bit unsigned integer.
- arrayBuffer <ArrayBuffer> | <SharedArrayBuffer> An ArrayBuffer instance.

Marks an ArrayBuffer as havings its contents transferred out of band. Pass the corresponding ArrayBuffer in the serializing context to serializer.transferArrayBuffer() (or return the id from serializer._getSharedArrayBufferId() in the case of SharedArrayBuffer s).

deserializer.getWireFormatVersion()

- Returns: <integer>

Reads the underlying wire format version. Likely mostly to be useful to legacy code reading old wire format versions. May not be called before .readHeader() .

deserializer.readUint32()

- Returns: <integer>

Read a raw 32-bit unsigned integer and return it. For use inside of a custom deserializer._readHostObject() .

deserializer.readUint64()

- Returns: <Array>

Read a raw 64-bit unsigned integer and return it as an array `[hi, lo]` with two 32-bit unsigned integer entries. For use inside of a custom deserializer._readHostObject() .

deserializer.readDouble()

- Returns: <number>

Read a JS `number` value. For use inside of a custom deserializer._readHostObject() .

deserializer.readRawBytes(length)

- Returns: <Buffer>

Read raw bytes from the deserializer's internal buffer. The `length` parameter must correspond to the length of the buffer that was passed to serializer.writeRawBytes() . For use inside of a custom deserializer._readHostObject() .

deserializer._readHostObject()

This method is called to read some kind of host object, i.e. an object that is created by native C++ bindings. If it is not possible to deserialize the data, a suitable exception should be thrown.

This method is not present on the `Deserializer` class itself but can be provided by subclasses.

class: v8.DefaultSerializer

A subclass of Serializer that serializes `TypedArray` (in particular Buffer) and `DataView` objects as host objects, and only stores the part of their underlying `ArrayBuffer` s that they are referring to.

class: v8.DefaultDeserializer

A subclass of Deserializer corresponding to the format written by DefaultSerializer .

VM (Executing JavaScript)

Stability: 2 - Stable

The `vm` module provides APIs for compiling and running code within V8 Virtual Machine contexts.

JavaScript code can be compiled and run immediately or compiled, saved, and run later.

A common use case is to run the code in a sandboxed environment. The sandboxed code uses a different V8 Context, meaning that it has a different global object than the rest of the code.

One can provide the context by "contextifying" a sandbox object. The sandboxed code treats any property on the sandbox like a global variable. Any changes on global variables caused by the sandboxed code are reflected in the sandbox object.

```
const vm = require('vm');

const x = 1;

const sandbox = { x: 2 };
vm.createContext(sandbox); // Contextify the sandbox.

const code = 'x += 40; var y = 17;';
// x and y are global variables in the sandboxed environment.
// Initially, x has the value 2 because that is the value of sandbox.x.
vm.runInContext(code, sandbox);

console.log(sandbox.x); // 42
console.log(sandbox.y); // 17

console.log(x); // 1; y is not defined.
```

The vm module is not a security mechanism. Do not use it to run untrusted code.

Class: vm.Module

Added in: v9.6.0

Stability: 1 - Experimental

This feature is only available with the `--experimental-vm-modules` *command flag enabled.*

The `vm.Module` class provides a low-level interface for using ECMAScript modules in VM contexts. It is the counterpart of the `vm.Script` class that closely mirrors Source Text Module Record s as defined in the ECMAScript specification.

Unlike `vm.Script` however, every `vm.Module` object is bound to a context from its creation. Operations on `vm.Module` objects are intrinsically asynchronous, in contrast with the synchronous nature of `vm.Script` objects. With the help of async functions, however, manipulating `vm.Module` objects is fairly straightforward.

Using a `vm.Module` object requires four distinct steps: creation/parsing, linking, instantiation, and evaluation. These four steps are illustrated in the following example.

This implementation lies at a lower level than the ECMAScript Module loader . There is also currently no way to interact with the Loader, though support is planned.

```
const vm = require('vm');

const contextifiedSandbox = vm.createContext({ secret: 42 });

(async () => {
  // Step 1
  //
  // Create a Module by constructing a new `vm.Module` object. This parses the
  // provided source text, throwing a `SyntaxError` if anything goes wrong. By
  // default, a Module is created in the top context. But here, we specify
  // `contextifiedSandbox` as the context this Module belongs to.
  //
  // Here, we attempt to obtain the default export from the module "foo", and
  // put it into local binding "secret".

  const bar = new vm.Module(`
    import s from 'foo';
    s;
```

```
  s;
`, { context: contextifiedSandbox });

// Step 2
//
// "Link" the imported dependencies of this Module to it.
//
// The provided linking callback (the "linker") accepts two arguments: the
// parent module (`bar` in this case) and the string that is the specifier of
// the imported module. The callback is expected to return a Module that
// corresponds to the provided specifier, with certain requirements documented
// in `module.link()`.
//
// If linking has not started for the returned Module, the same linker
// callback will be called on the returned Module.
//
// Even top-level Modules without dependencies must be explicitly linked. The
// callback provided would never be called, however.
//
// The link() method returns a Promise that will be resolved when all the
// Promises returned by the linker resolve.
//
// Note: This is a contrived example in that the linker function creates a new
// "foo" module every time it is called. In a full-fledged module system, a
// cache would probably be used to avoid duplicated modules.

async function linker(specifier, referencingModule) {
  if (specifier === 'foo') {
    return new vm.Module(`
      // The "secret" variable refers to the global variable we added to
      // "contextifiedSandbox" when creating the context.
      export default secret;
    `, { context: referencingModule.context });

    // Using `contextifiedSandbox` instead of `referencingModule.context`
    // here would work as well.
  }
  throw new Error(`Unable to resolve dependency: ${specifier}`);
}
await bar.link(linker);

// Step 3
//
// Instantiate the top-level Module.
//
// Only the top-level Module needs to be explicitly instantiated; its
// dependencies will be recursively instantiated by instantiate().

bar.instantiate();

// Step 4
//
// Evaluate the Module. The evaluate() method returns a Promise with a single
// property "result" that contains the result of the very last statement
// executed in the Module. In the case of `bar`, it is `s;`, which refers to
// the default export of the `foo` module, the `secret` we set in the
// beginning to 42.

const { result } = await bar.evaluate();
```

```
  const { result } = await bar.evaluate();

  console.log(result);
  // Prints 42.
})();
```

Constructor: new vm.Module(code[, options])

- `code` <string> JavaScript Module code to parse
- `options`
  - `url` <string> URL used in module resolution and stack traces. **Default**: `'vm:module(i)'` where `i` is a context-specific ascending index.
  - `context` <Object> The contextified object as returned by the `vm.createContext()` method, to compile and evaluate this Module in.
  - `lineOffset` <integer> Specifies the line number offset that is displayed in stack traces produced by this Module.
  - `columnOffset` <integer> Spcifies the column number offset that is displayed in stack traces produced by this Module.

Creates a new ES `Module` object.

module.dependencySpecifiers

- <string[]>

The specifiers of all dependencies of this module. The returned array is frozen to disallow any changes to it.

Corresponds to the [[RequestedModules]] field of Source Text Module Record s in the ECMAScript specification.

module.error

- <any>

If the `module.status` is `'errored'`, this property contains the exception thrown by the module during evaluation. If the status is anything else, accessing this property will result in a thrown exception.

The value `undefined` cannot be used for cases where there is not a thrown exception due to possible ambiguity with `throw undefined;`.

Corresponds to the [[EvaluationError]] field of Source Text Module Record s in the ECMAScript specification.

module.linkingStatus

- <string>

The current linking status of `module`. It will be one of the following values:

- `'unlinked'`: `module.link()` has not yet been called.
- `'linking'`: `module.link()` has been called, but not all Promises returned by the linker function have been resolved yet.
- `'linked'`: `module.link()` has been called, and all its dependencies have been successfully linked.
- `'errored'`: `module.link()` has been called, but at least one of its dependencies failed to link, either because the callback returned a Promise that is rejected, or because the Module the callback returned is invalid.

module.namespace

- <Object>

The namespace object of the module. This is only available after instantiation (`module.instantiate()`) has completed.

Corresponds to the GetModuleNamespace abstract operation in the ECMAScript specification.

module.status

- <string>

The current status of the module. Will be one of:

- `'uninstantiated'`: The module is not instantiated. It may because of any of the following reasons:
  - The module was just created.
  - `module.instantiate()` has been called on this module, but it failed for some reason.

  This status does not convey any information regarding if `module.link()` has been called. See `module.linkingStatus` for that.
- `'instantiating'`: The module is currently being instantiated through a `module.instantiate()` call on itself or a parent module.
- `'instantiated'`: The module has been instantiated successfully, but `module.evaluate()` has not yet been called.
- `'evaluating'`: The module is being evaluated through a `module.evaluate()` on itself or a parent module.
- `'evaluated'`: The module has been successfully evaluated.
- `'errored'`: The module has been evaluated, but an exception was thrown.

Other than `'errored'`, this status string corresponds to the specification's Source Text Module Record 's [[Status]] field. `'errored'` corresponds to `'evaluated'` in the specification, but with [[EvaluationError]] set to a value that is not `undefined`.

module.url

- <string>

The URL of the current module, as set in the constructor.

module.evaluate(options)

- `options` <Object>
  - `timeout` <number> Specifies the number of milliseconds to evaluate before terminating execution. If execution is interrupted, an Error will be thrown.
  - `breakOnSigint` <boolean> If `true`, the execution will be terminated when `SIGINT` (Ctrl+C) is received. Existing handlers for the event that have been attached via `process.on('SIGINT')` will be disabled during script execution, but will continue to work after that. If execution is interrupted, an Error will be thrown.
- Returns: <Promise>

Evaluate the module.

This must be called after the module has been instantiated; otherwise it will throw an error. It could be called also when the module has already been evaluated, in which case it will do one of the following two things:

- return `undefined` if the initial evaluation ended in success (`module.status` is `'evaluated'`)
- rethrow the same exception the initial evaluation threw if the initial evaluation ended in an error (`module.status` is `'errored'`)

This method cannot be called while the module is being evaluated (`module.status` is `'evaluating'`) to prevent infinite recursion.

Corresponds to the Evaluate() concrete method field of Source Text Module Record s in the ECMAScript specification.

module.instantiate()

Instantiate the module. This must be called after linking has completed (`linkingStatus` is `'linked'`); otherwise it will throw an error. It may also throw an exception if one of the dependencies does not provide an export the parent module requires.

However, if this function succeeded, further calls to this function after the initial instantiation will be no-ops, to be consistent with the ECMAScript specification.

Unlike other methods operating on `Module`, this function completes synchronously and returns nothing.

Corresponds to the Instantiate() concrete method field of Source Text Module Record s in the ECMAScript specification.

module.link(linker)

- `linker` <Function>
- Returns: <Promise>

Link module dependencies. This method must be called before instantiation, and can only be called once per module.

Two parameters will be passed to the `linker` function:

- `specifier` The specifier of the requested module:

```
import foo from 'foo';
//              ^^^^^ the module specifier
```

- `referencingModule` The `Module` object `link()` is called on.

The function is expected to return a `Module` object or a `Promise` that eventually resolves to a `Module` object. The returned `Module` must satisfy the following two invariants:

- It must belong to the same context as the parent `Module`.
- Its `linkingStatus` must not be `'errored'`.

If the returned `Module`'s `linkingStatus` is `'unlinked'`, this method will be recursively called on the returned `Module` with the same provided `linker` function.

`link()` returns a `Promise` that will either get resolved when all linking instances resolve to a valid `Module`, or rejected if the linker function either throws an exception or returns an invalid `Module`.

The linker function roughly corresponds to the implementation-defined HostResolveImportedModule abstract operation in the ECMAScript specification, with a few key differences:

- The linker function is allowed to be asynchronous while HostResolveImportedModule is synchronous.
- The linker function is executed during linking, a Node.js-specific stage before instantiation, while HostResolveImportedModule is called during instantiation.

The actual HostResolveImportedModule implementation used during module instantiation is one that returns the modules linked during linking. Since at that point all modules would have been fully linked already, the HostResolveImportedModule implementation is fully synchronous per specification.

Class: vm.Script

Added in: v0.3.1

Instances of the `vm.Script` class contain precompiled scripts that can be executed in specific sandboxes (or "contexts").

new vm.Script(code, options)

▸ History

- `code` <string> The JavaScript code to compile.
- `options`
  - `filename` <string> Specifies the filename used in stack traces produced by this script.
  - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
  - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
  - `displayErrors` <boolean> When `true`, if an Error error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
  - `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an Error will be thrown.
  - `cachedData` <Buffer> Provides an optional `Buffer` with V8's code cache data for the supplied source. When supplied, the `cachedDataRejected` value will be set to either `true` or `false` depending on acceptance of the data by V8.
  - `produceCachedData` <boolean> When `true` and no `cachedData` is present, V8 will attempt to produce code cache data for `code`. Upon success, a `Buffer` with V8's code cache data will be produced and stored in the `cachedData` property of the returned `vm.Script` instance. The `cachedDataProduced` value will be set to either `true` or `false` depending on whether code cache data is produced successfully.

Creating a new `vm.Script` object compiles `code` but does not run it. The compiled `vm.Script` can be run later multiple times. The `code` is not

bound to any global object; rather, it is bound before each run, just for that run.

script.runInContext(contextifiedSandbox[, options])

▶ History

- `contextifiedSandbox` <Object> A contextified object as returned by the `vm.createContext()` method.
- `options` <Object>
  - `filename` <string> Specifies the filename used in stack traces produced by this script.
  - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
  - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
  - `displayErrors` <boolean> When `true`, if an Error error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
  - `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an Error will be thrown.
  - `breakOnSigint`: if `true`, the execution will be terminated when `SIGINT` (Ctrl+C) is received. Existing handlers for the event that have been attached via `process.on('SIGINT')` will be disabled during script execution, but will continue to work after that. If execution is terminated, an Error will be thrown.

Runs the compiled code contained by the `vm.Script` object within the given `contextifiedSandbox` and returns the result. Running code does not have access to local scope.

The following example compiles code that increments a global variable, sets the value of another global variable, then execute the code multiple times. The globals are contained in the `sandbox` object.

```
const util = require('util');
const vm = require('vm');

const sandbox = {
  animal: 'cat',
  count: 2
};

const script = new vm.Script('count += 1; name = "kitty";');

const context = vm.createContext(sandbox);
for (let i = 0; i < 10; ++i) {
  script.runInContext(context);
}

console.log(util.inspect(sandbox));

// { animal: 'cat', count: 12, name: 'kitty' }
```

Using the `timeout` or `breakOnSigint` options will result in new event loops and corresponding threads being started, which have a non-zero performance overhead.

script.runInNewContext([sandbox[, options]])

Added in: v0.3.1

- `sandbox` <Object> An object that will be contextified. If `undefined`, a new object will be created.
- `options` <Object>
  - `filename` <string> Specifies the filename used in stack traces produced by this script.
  - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
  - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
  - `displayErrors` <boolean> When `true`, if an Error error occurs while compiling the `code`, the line of code causing the error is attached

to the stack trace.

- `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an Error will be thrown.
- `contextName` <string> Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
- `contextOrigin` <string> Origin corresponding to the newly created context for display purposes. The origin should be formatted like a URL, but with only the scheme, host, and port (if necessary), like the value of the url.origin property of a URL object. Most notably, this string should omit the trailing slash, as that denotes a path. **Default:** `''`.

First contextifies the given `sandbox`, runs the compiled code contained by the `vm.Script` object within the created sandbox, and returns the result. Running code does not have access to local scope.

The following example compiles code that sets a global variable, then executes the code multiple times in different contexts. The globals are set on and contained within each individual `sandbox`.

```
const util = require('util');
const vm = require('vm');

const script = new vm.Script('globalVar = "set"');

const sandboxes = [{}, {}, {}];
sandboxes.forEach((sandbox) => {
  script.runInNewContext(sandbox);
});

console.log(util.inspect(sandboxes));

// [{ globalVar: 'set' }, { globalVar: 'set' }, { globalVar: 'set' }]
```

script.runInThisContext(options)

Added in: v0.3.1

- `options` <Object>
  - `filename` <string> Specifies the filename used in stack traces produced by this script.
  - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
  - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
  - `displayErrors` <boolean> When `true`, if an Error error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
  - `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an Error will be thrown.

Runs the compiled code contained by the `vm.Script` within the context of the current `global` object. Running code does not have access to local scope, but *does* have access to the current `global` object.

The following example compiles code that increments a `global` variable then executes that code multiple times:

```
const vm = require('vm');

global.globalVar = 0;

const script = new vm.Script('globalVar += 1', { filename: 'myfile.vm' });

for (let i = 0; i < 1000; ++i) {
  script.runInThisContext();
}

console.log(globalVar);

// 1000
```

vm.createContext([sandbox[, options]])

Added in: v0.3.1

- `sandbox` <Object>
- `options` <Object>
  - `name` <string> Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
  - `origin` <string> Origin corresponding to the newly created context for display purposes. The origin should be formatted like a URL, but with only the scheme, host, and port (if necessary), like the value of the `url.origin` property of a `URL` object. Most notably, this string should omit the trailing slash, as that denotes a path. **Default:** `''`.

If given a `sandbox` object, the `vm.createContext()` method will prepare that sandbox so that it can be used in calls to `vm.runInContext()` or `script.runInContext()`. Inside such scripts, the `sandbox` object will be the global object, retaining all of its existing properties but also having the built-in objects and functions any standard global object has. Outside of scripts run by the vm module, global variables will remain unchanged.

```
const util = require('util');
const vm = require('vm');

global.globalVar = 3;

const sandbox = { globalVar: 1 };
vm.createContext(sandbox);

vm.runInContext('globalVar *= 2;', sandbox);

console.log(util.inspect(sandbox)); // { globalVar: 2 }

console.log(util.inspect(globalVar)); // 3
```

If `sandbox` is omitted (or passed explicitly as `undefined`), a new, empty contextified sandbox object will be returned.

The `vm.createContext()` method is primarily useful for creating a single sandbox that can be used to run multiple scripts. For instance, if emulating a web browser, the method can be used to create a single sandbox representing a window's global object, then run all `<script>` tags together within the context of that sandbox.

The provided `name` and `origin` of the context are made visible through the Inspector API.

vm.isContext(sandbox)

Added in: v0.11.7

- `sandbox` <Object>

Returns `true` if the given `sandbox` object has been contextified using `vm.createContext()` .

vm.runInContext(code, contextifiedSandbox[, options])

- `code` <string> The JavaScript code to compile and run.
- `contextifiedSandbox` <Object> The contextified object that will be used as the `global` when the `code` is compiled and run.
- `options`
  - `filename` <string> Specifies the filename used in stack traces produced by this script.
  - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
  - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
  - `displayErrors` <boolean> When `true` , if an Error error occurs while compiling the `code` , the line of code causing the error is attached to the stack trace.
  - `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an Error will be thrown.

The `vm.runInContext()` method compiles `code` , runs it within the context of the `contextifiedSandbox` , then returns the result. Running code does not have access to the local scope. The `contextifiedSandbox` object *must* have been previously contextified using the `vm.createContext()` method.

The following example compiles and executes different scripts using a single contextified object:

```
const util = require('util');
const vm = require('vm');

const sandbox = { globalVar: 1 };
vm.createContext(sandbox);

for (let i = 0; i < 10; ++i) {
  vm.runInContext('globalVar *= 2;', sandbox);
}
console.log(util.inspect(sandbox));

// { globalVar: 1024 }
```

vm.runInDebugContext(code)

▶ History

Stability: 0 - Deprecated. An alternative is in development.

- `code` <string> The JavaScript code to compile and run.

The `vm.runInDebugContext()` method compiles and executes `code` inside the V8 debug context. The primary use case is to gain access to the V8 `Debug` object:

```
const vm = require('vm');
const Debug = vm.runInDebugContext('Debug');
console.log(Debug.findScript(process.emit).name);  // 'events.js'
console.log(Debug.findScript(process.exit).name);  // 'internal/process.js'
```

Note: The debug context and object are intrinsically tied to V8's debugger implementation and may change (or even be removed) without prior warning.

The `Debug` object can also be made available using the V8-specific `--expose_debug_as=` command line option .

vm.runInNewContext(code[, sandbox][, options])

Added in: v0.3.1

- `code` <string> The JavaScript code to compile and run.
- `sandbox` <Object> An object that will be contextified. If `undefined`, a new object will be created.
- `options`
  - `filename` <string> Specifies the filename used in stack traces produced by this script.
  - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
  - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
  - `displayErrors` <boolean> When `true`, if an Error error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
  - `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an Error will be thrown.
  - `contextName` <string> Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
  - `contextOrigin` <string> Origin corresponding to the newly created context for display purposes. The origin should be formatted like a URL, but with only the scheme, host, and port (if necessary), like the value of the url.origin property of a URL object. Most notably, this string should omit the trailing slash, as that denotes a path. **Default:** `''`.

The `vm.runInNewContext()` first contextifies the given `sandbox` object (or creates a new `sandbox` if passed as `undefined`), compiles the `code`, runs it within the context of the created context, then returns the result. Running code does not have access to the local scope.

The following example compiles and executes code that increments a global variable and sets a new one. These globals are contained in the `sandbox`.

```
const util = require('util');
const vm = require('vm');

const sandbox = {
  animal: 'cat',
  count: 2
};

vm.runInNewContext('count += 1; name = "kitty"', sandbox);
console.log(util.inspect(sandbox));

// { animal: 'cat', count: 3, name: 'kitty' }
```

vm.runInThisContext(code[, options])

Added in: v0.3.1

- `code` <string> The JavaScript code to compile and run.
- `options`
  - `filename` <string> Specifies the filename used in stack traces produced by this script.
  - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
  - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
  - `displayErrors` <boolean> When `true`, if an Error error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
  - `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an Error will be thrown.

`vm.runInThisContext()` compiles `code`, runs it within the context of the current `global` and returns the result. Running code does not have access to local scope, but does have access to the current `global` object.

The following example illustrates using both `vm.runInThisContext()` and the JavaScript eval() function to run the same code:

```js
const vm = require('vm');
let localVar = 'initial value';

const vmResult = vm.runInThisContext('localVar = "vm";');
console.log('vmResult:', vmResult);
console.log('localVar:', localVar);

const evalResult = eval('localVar = "eval";');
console.log('evalResult:', evalResult);
console.log('localVar:', localVar);

// vmResult: 'vm', localVar: 'initial value'
// evalResult: 'eval', localVar: 'eval'
```

Because `vm.runInThisContext()` does not have access to the local scope, `localVar` is unchanged. In contrast, `eval()` *does* have access to the local scope, so the value `localVar` is changed. In this way `vm.runInThisContext()` is much like an indirect `eval()` call, e.g. `(0,eval)('code')`.

Example: Running an HTTP Server within a VM

When using either `script.runInThisContext()` or `vm.runInThisContext()`, the code is executed within the current V8 global context. The code passed to this VM context will have its own isolated scope.

In order to run a simple web server using the `http` module the code passed to the context must either call `require('http')` on its own, or have a reference to the `http` module passed to it. For instance:

```js
'use strict';
const vm = require('vm');

const code = `
((require) => {
  const http = require('http');

  http.createServer((request, response) => {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.end('Hello World\\n');
  }).listen(8124);

  console.log('Server running at http://127.0.0.1:8124/');
})`;

vm.runInThisContext(code)(require);
```

The `require()` in the above case shares the state with the context it is passed from. This may introduce risks when untrusted code is executed, e.g. altering objects in the context in unwanted ways.

What does it mean to "contextify" an object?

All JavaScript executed within Node.js runs within the scope of a "context". According to the V8 Embedder's Guide:

> In V8, a context is an execution environment that allows separate, unrelated, JavaScript applications to run in a single instance of V8. You must explicitly specify the context in which you want any JavaScript code to be run.

When the method `vm.createContext()` is called, the `sandbox` object that is passed in (or a newly created object if `sandbox` is `undefined`) is associated internally with a new instance of a V8 Context. This V8 Context provides the `code` run using the `vm` module's methods with an isolated global environment within which it can operate. The process of creating the V8 Context and associating it with the `sandbox` object is what this document refers to as "contextifying" the `sandbox`.

Zlib

Stability: 2 - Stable

The `zlib` module provides compression functionality implemented using Gzip and Deflate/Inflate. It can be accessed using:

```
const zlib = require('zlib');
```

Compressing or decompressing a stream (such as a file) can be accomplished by piping the source stream data through a `zlib` stream into a destination stream:

```
const gzip = zlib.createGzip();
const fs = require('fs');
const inp = fs.createReadStream('input.txt');
const out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

It is also possible to compress or decompress data in a single step:

```
const input = '.................................';
zlib.deflate(input, (err, buffer) => {
  if (!err) {
    console.log(buffer.toString('base64'));
  } else {
    // handle error
  }
});

const buffer = Buffer.from('eJzT0yMAAGTvBe8=', 'base64');
zlib.unzip(buffer, (err, buffer) => {
  if (!err) {
    console.log(buffer.toString());
  } else {
    // handle error
  }
});
```

Threadpool Usage

Note that all zlib APIs except those that are explicitly synchronous use libuv's threadpool, which can have surprising and negative performance implications for some applications, see the UV_THREADPOOL_SIZE documentation for more information.

Compressing HTTP requests and responses

The `zlib` module can be used to implement support for the `gzip` and `deflate` content-encoding mechanisms defined by HTTP .

The HTTP Accept-Encoding header is used within an http request to identify the compression encodings accepted by the client. The Content-Encoding header is used to identify the compression encodings actually applied to a message.

The examples given below are drastically simplified to show the basic concept. Using `zlib` encoding can be expensive, and the results ought to be cached. See Memory Usage Tuning for more information on the speed/memory/compression tradeoffs involved in `zlib` usage.

```js
// client request example
const zlib = require('zlib');
const http = require('http');
const fs = require('fs');
const request = http.get({ host: 'example.com',
                           path: '/',
                           port: 80,
                           headers: { 'Accept-Encoding': 'gzip,deflate' } });
request.on('response', (response) => {
  const output = fs.createWriteStream('example.com_index.html');

  switch (response.headers['content-encoding']) {
    // or, just use zlib.createUnzip() to handle both cases
    case 'gzip':
      response.pipe(zlib.createGunzip()).pipe(output);
      break;
    case 'deflate':
      response.pipe(zlib.createInflate()).pipe(output);
      break;
    default:
      response.pipe(output);
      break;
  }
});
```

```js
// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
const zlib = require('zlib');
const http = require('http');
const fs = require('fs');
http.createServer((request, response) => {
  const raw = fs.createReadStream('index.html');
  let acceptEncoding = request.headers['accept-encoding'];
  if (!acceptEncoding) {
    acceptEncoding = '';
  }

  // Note: This is not a conformant accept-encoding parser.
  // See https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3
  if (/\bdeflate\b/.test(acceptEncoding)) {
    response.writeHead(200, { 'Content-Encoding': 'deflate' });
    raw.pipe(zlib.createDeflate()).pipe(response);
  } else if (/\bgzip\b/.test(acceptEncoding)) {
    response.writeHead(200, { 'Content-Encoding': 'gzip' });
    raw.pipe(zlib.createGzip()).pipe(response);
  } else {
    response.writeHead(200, {});
    raw.pipe(response);
  }
}).listen(1337);
```

By default, the `zlib` methods will throw an error when decompressing truncated data. However, if it is known that the data is incomplete, or the desire is to inspect only the beginning of a compressed file, it is possible to suppress the default error handling by changing the flushing method that is used to decompress the last chunk of input data:

```
// This is a truncated version of the buffer from the above examples
const buffer = Buffer.from('eJzT0yMA', 'base64');

zlib.unzip(
  buffer,
  { finishFlush: zlib.constants.Z_SYNC_FLUSH },
  (err, buffer) => {
    if (!err) {
      console.log(buffer.toString());
    } else {
      // handle error
    }
  });
```

This will not change the behavior in other error-throwing situations, e.g. when the input data has an invalid format. Using this method, it will not be possible to determine whether the input ended prematurely or lacks the integrity checks, making it necessary to manually check that the decompressed result is valid.

Memory Usage Tuning

From `zlib/zconf.h`, modified to node.js's usage:

The memory requirements for deflate are (in bytes):

```
(1 << (windowBits + 2)) + (1 << (memLevel + 9))
```

That is: 128K for windowBits = 15 + 128K for memLevel = 8 (default values) plus a few kilobytes for small objects.

For example, to reduce the default memory requirements from 256K to 128K, the options should be set to:

```
const options = { windowBits: 14, memLevel: 7 };
```

This will, however, generally degrade compression.

The memory requirements for inflate are (in bytes) `1 << windowBits`. That is, 32K for windowBits = 15 (default value) plus a few kilobytes for small objects.

This is in addition to a single internal output slab buffer of size `chunkSize`, which defaults to 16K.

The speed of `zlib` compression is affected most dramatically by the `level` setting. A higher level will result in better compression, but will take longer to complete. A lower level will result in less compression, but will be much faster.

In general, greater memory usage options will mean that Node.js has to make fewer calls to `zlib` because it will be able to process more data on each `write` operation. So, this is another factor that affects the speed, at the cost of memory usage.

Flushing

Calling `.flush()` on a compression stream will make `zlib` return as much output as currently possible. This may come at the cost of degraded compression quality, but can be useful when data needs to be available as soon as possible.

In the following example, `flush()` is used to write a compressed partial HTTP response to the client:

```
const zlib = require('zlib');
const http = require('http');

http.createServer((request, response) => {
  // For the sake of simplicity, the Accept-Encoding checks are omitted.
  response.writeHead(200, { 'content-encoding': 'gzip' });
  const output = zlib.createGzip();
  output.pipe(response);

  setInterval(() => {
    output.write(`The current time is ${Date()}\n`, () => {
      // The data has been passed to zlib, but the compression algorithm may
      // have decided to buffer the data for more efficient compression.
      // Calling .flush() will make the data available as soon as the client
      // is ready to receive it.
      output.flush();
    });
  }, 1000);
}).listen(1337);
```

Constants

Added in: v0.5.8

All of the constants defined in `zlib.h` are also defined on `require('zlib').constants`. In the normal course of operations, it will not be necessary to use these constants. They are documented so that their presence is not surprising. This section is taken almost directly from the zlib documentation. See https://zlib.net/manual.html#Constants for more details.

Previously, the constants were available directly from `require('zlib')`, for instance `zlib.Z_NO_FLUSH`. Accessing the constants directly from the module is currently still possible but is deprecated.

Allowed flush values.

- `zlib.constants.Z_NO_FLUSH`
- `zlib.constants.Z_PARTIAL_FLUSH`
- `zlib.constants.Z_SYNC_FLUSH`
- `zlib.constants.Z_FULL_FLUSH`
- `zlib.constants.Z_FINISH`
- `zlib.constants.Z_BLOCK`
- `zlib.constants.Z_TREES`

Return codes for the compression/decompression functions. Negative values are errors, positive values are used for special but normal events.

- `zlib.constants.Z_OK`
- `zlib.constants.Z_STREAM_END`
- `zlib.constants.Z_NEED_DICT`
- `zlib.constants.Z_ERRNO`
- `zlib.constants.Z_STREAM_ERROR`
- `zlib.constants.Z_DATA_ERROR`
- `zlib.constants.Z_MEM_ERROR`
- `zlib.constants.Z_BUF_ERROR`
- `zlib.constants.Z_VERSION_ERROR`

Compression levels.

- zlib.constants.Z_NO_COMPRESSION
- zlib.constants.Z_BEST_SPEED
- zlib.constants.Z_BEST_COMPRESSION
- zlib.constants.Z_DEFAULT_COMPRESSION

Compression strategy.

- zlib.constants.Z_FILTERED
- zlib.constants.Z_HUFFMAN_ONLY
- zlib.constants.Z_RLE
- zlib.constants.Z_FIXED
- zlib.constants.Z_DEFAULT_STRATEGY

Class Options

▸ History

Each class takes an `options` object. All options are optional.

Note that some options are only relevant when compressing, and are ignored by the decompression classes.

- `flush` <integer> (default: `zlib.constants.Z_NO_FLUSH`)
- `finishFlush` <integer> (default: `zlib.constants.Z_FINISH`)
- `chunkSize` <integer> (default: 16*1024)
- `windowBits` <integer>
- `level` <integer> (compression only)
- `memLevel` <integer> (compression only)
- `strategy` <integer> (compression only)
- `dictionary` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> (deflate/inflate only, empty dictionary by default)
- `info` <boolean> (If `true`, returns an object with `buffer` and `engine`)

See the description of `deflateInit2` and `inflateInit2` at https://zlib.net/manual.html#Advanced for more information on these.

Class: zlib.Deflate

Added in: v0.5.8

Compress data using deflate.

Class: zlib.DeflateRaw

Added in: v0.5.8

Compress data using deflate, and do not append a `zlib` header.

Class: zlib.Gunzip

▸ History

Decompress a gzip stream.

Class: zlib.Gzip

Added in: v0.5.8

Compress data using gzip.

Class: zlib.Inflate

▸ History

Decompress a deflate stream.

Class: zlib.InflateRaw

▸ History

Decompress a raw deflate stream.

Class: zlib.Unzip

Added in: v0.5.8

Decompress either a Gzip- or Deflate-compressed stream by auto-detecting the header.

Class: zlib.Zlib

Added in: v0.5.8

Not exported by the `zlib` module. It is documented here because it is the base class of the compressor/decompressor classes.

zlib.bytesRead

Added in: v8.1.0

- <number>

The `zlib.bytesRead` property specifies the number of bytes read by the engine before the bytes are processed (compressed or decompressed, as appropriate for the derived class).

zlib.close([callback])

Added in: v0.9.4

Close the underlying handle.

zlib.flush([kind], callback)

Added in: v0.5.8

`kind` defaults to `zlib.constants.Z_FULL_FLUSH`.

Flush pending data. Don't call this frivolously, premature flushes negatively impact the effectiveness of the compression algorithm.

Calling this only flushes data from the internal `zlib` state, and does not perform flushing of any kind on the streams level. Rather, it behaves like a normal call to `.write()`, i.e. it will be queued up behind other pending writes and will only produce output when data is being read from the stream.

zlib.params(level, strategy, callback)

Added in: v0.11.4

Dynamically update the compression level and compression strategy. Only applicable to deflate algorithm.

zlib.reset()

Added in: v0.7.0

Reset the compressor/decompressor to factory defaults. Only applicable to the inflate and deflate algorithms.

zlib.constants

Added in: v7.0.0

Provides an object enumerating Zlib-related constants.

zlib.createDeflate(options)

Added in: v0.5.8

Creates and returns a new Deflate object with the given options.

zlib.createDeflateRaw(options)

Added in: v0.5.8

Creates and returns a new DeflateRaw object with the given options.

An upgrade of zlib from 1.2.8 to 1.2.11 changed behavior when windowBits is set to 8 for raw deflate streams. zlib would automatically set windowBits to 9 if was initially set to 8. Newer versions of zlib will throw an exception, so Node.js restored the original behavior of upgrading a value of 8 to 9, since passing `windowBits = 9` to zlib actually results in a compressed stream that effectively uses an 8-bit window only.

zlib.createGunzip(options)

Added in: v0.5.8

Creates and returns a new Gunzip object with the given options.

zlib.createGzip(options)

Added in: v0.5.8

Creates and returns a new Gzip object with the given options.

zlib.createInflate(options)

Added in: v0.5.8

Creates and returns a new Inflate object with the given options.

zlib.createInflateRaw(options)

Added in: v0.5.8

Creates and returns a new InflateRaw object with the given options.

zlib.createUnzip(options)

Added in: v0.5.8

Creates and returns a new Unzip object with the given options.

Convenience Methods

All of these take a Buffer, TypedArray, DataView, ArrayBuffer or string as the first argument, an optional second argument to supply options to the `zlib` classes and will call the supplied callback with `callback(error, result)`.

Every method has a `*Sync` counterpart, which accept the same arguments, but without a callback.

zlib.deflate(buffer[, options], callback)

▸ History

zlib.deflateSync(buffer[, options])

▸ History

- `buffer` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>

Compress a chunk of data with Deflate.

zlib.deflateRaw(buffer[, options], callback)

▸ History

zlib.deflateRawSync(buffer[, options])

▸ History

- `buffer` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>

Compress a chunk of data with DeflateRaw.

zlib.gunzip(buffer[, options], callback)

▸ History

zlib.gunzipSync(buffer[, options])

▸ History

- `buffer` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>

Decompress a chunk of data with Gunzip.

zlib.gzip(buffer[, options], callback)

▸ History

zlib.gzipSync(buffer[, options])

▸ History

- `buffer` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>

Compress a chunk of data with Gzip.

zlib.inflate(buffer[, options], callback)

▸ History

zlib.inflateSync(buffer[, options])

▸ History

- `buffer` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>

Decompress a chunk of data with Inflate.

zlib.inflateRaw(buffer[, options], callback)

▸ History

zlib.inflateRawSync(buffer[, options])

▶ History

- `buffer` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>

Decompress a chunk of data with InflateRaw .

zlib.unzip(buffer[, options], callback)

▶ History

zlib.unzipSync(buffer[, options])

▶ History

- `buffer` <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>

Decompress a chunk of data with Unzip .